Universidad Complutense de Madrid

Facultad de Informática

Predicción de series temporales con k-NN sobre Spark

Alumnos **Sebastián Águila Sánchez**
Sergio Montero Cobo de Guzmán

Director del proyecto **Javier Arroyo Gallardo**

Codirector del proyecto **Albert Meco Alías**

Trabajo de Fin de Grado

Ingeniería de Software

Septiembre de 2018

Índice general

Palabras clave

Palabras clave en Español

- Regresión
- Predicción
- k-NN
- Series temporales
- Spark
- RDD

Keywords in English

- Regression
- Prediction
- k-NN
- Time Series
- Spark
- RDD

Resumen

Resumen en Español

Este trabajo ha tenido como objetivo la implementación de un algoritmo de predicción k-NN para series temporales, haciendo uso del paradigma de procesamiento distribuido Spark, con el fin conseguir una solución Big Data.
Conseguir esta solución es muy interesante, puesto que es un algoritmo de aprendizaje basado en instancias no genera un modelo para realizar las predicciones, sino que se apoya de un histórico.
Un histórico lo suficientemente grande podría desbordar la memoria de cualquier computador.
Con el uso del algoritmo distribuido, se puede procesar sin limitaciones este histórico por muy grande que sea, permitiendo tener en cuenta más datos para componer la solución, obteniendo mejores predicciones que las que se obtendrían usando una solución *mono-máquina* con un histórico limitado.

Summary in English

The objective of this work was to implement a k-NN forecasting algorithm for time series, using the distributed processing paradigm Spark, in order to achieve a Big Data solution.
This solution is very useful, since it's an instance based learning algorithm and it does not generate a model to make predictions, instead it uses historical data.
With enough historical data, it is possible to overflow the memory of any computer..
With the use of the distributed algorithm, this historical can be processed without limitations, no matter how big it is, allowing to have more data to compose the solution, obtaining better predictions than what would be obtained using a *mono-machine* solution with a limited historical.

Capítulo 1

Introducción

La búsqueda de la capacidad para explotar correctamente los datos con el fin de sacarles un valor presenta un grave problema, la gran cantidad y diversidad de datos que tenemos al alcance hacen imposible su procesamiento haciendo uso de la computación tradicional *mono-máquina*, debido a que superan la **capacidad de computación y memoria** de un cualquier computador, por muy potente que sea.

1.1. Importancia de las soluciones Big Data en algoritmos de aprendizaje automático

Dado un escenario en el que sea de vital importancia procesar la mayor cantidad de datos posibles, se manifiesta la necesidad de tener un sistema que emplee una computación distribuida con una **escalabilidad horizontal** sencilla y económica, que permita realizar el procesamiento de la totalidad de la información accesible con el fin de intentar sacarle valor.

Con el fin de intentar sacar valor a los datos, es habitual que se analicen con ***algoritmos de aprendizaje automático***, estos **permiten extraer comportamientos o tendencias** que realizando un buen uso de esta información **consiguen añadir valor adicional a los datos**.

En **un escenario Big Data** en el que hay infinidad de volumen de datos que necesitan ser procesados y adaptados con velocidad, es imposible usar estos algoritmos debido a que el escenario **sobrepasa la capacidad de computación y de memoria de la máquina que realiza el procesamiento**.

Este es el motivo por el cual los algoritmos de *aprendizaje automático* están siendo integrados con soluciones de computación distribuida en entornos Big Data, este tipo de soluciones **permiten mejorar las características de los algoritmos de aprendizaje automático, haciéndolos más robustos y potentes**, permitiendo procesar más instancias en las predicciones usando una cantidad ilimitada de ejemplos de entrenamiento. Al usar una computación distribuida el procesamiento **se paraleliza** en varias máquinas **consiguiendo generan las predicciones en el menor tiempo posible**.

Cualquier tipo de algoritmo de aprendizaje automático se puede beneficiar de la **capacidad de procesamiento y la paralelización** que ofrece esta tecnología, pero **solo tiene sentido aplicar este tipo de computación si hay una enorme cantidad de datos**.

1.2. Aprendizaje basado en instancias

Es un tipo de aprendizaje automático. Al contrario que la mayoría de algoritmos de aprendizaje automático, **no genera un modelo para realizar las predicciones, sino que usa todo el conjunto de datos conocidos o instancias para componer la solución**, por esto se dice que es ***memory-based learning***.

Este tipo de algoritmos **usan los datos o instancias como tabla de consulta para las predicciones**, revisando que la nueva instancia a predecir se encuentre en la tabla o teniendo en cuenta instancias similares.

Por esta razón cuanta más cantidad de datos se tengan, más viable es encontrar la instancia a predecir en la tabla de consulta, así como encontrar más instancias similares que afinen mejor las predicciones.

Es fundamental usar todos los datos disponibles con el fin de buscar las instancias que ajusten mejor la predicción o usar todas las posibles pero con una función de ponderación para realizar el ajuste.
Las **diferencias entre las variantes de los algoritmos de aprendizaje basado en instancias** se centran en:

- El **número determinado de instancias** que serán usadas para componer la solución.
- La **distancia** en la que se relaciona, la similitud entre las instancias y la nueva variable de entrada a predecir.
- La **composición de la solución** en la que se tiene en cuenta el número de instancias y el tipo de distancia a utilizar.

En cada predicción se procesan todo el conjunto de datos para elaborar la tabla de consulta, este procesamiento no se realiza hasta que no se tiene el dato sobre el que hay que hacer la predicción, es por esta razón que se le denomina lazy. Todo el procesamiento se centra en comparar la nueva instancia con todas las anteriores y esto no puede ocurrir hasta que no se tenga la nueva instancia y se quiera realizar la predicción.

1.3. Motivación

Como podemos extraer del apartado anterior **la efectividad de un algoritmo de *aprendizaje basado en instancias* esta determinada por la riqueza del conjunto de datos o instancias** y en gran medida en procesar la mayor cantidad de datos posibles a la hora de realizar las predicciones.

Este planteamiento presenta un grave inconveniente, debido a que **la capacidad de memoria de una máquina es limitada**, **no se puede procesar un volumen de datos ilimitado**. Este es el principal inconveniente de este tipo de algoritmos de aprendizaje automático usando una computación *mono-máquina*.

Si se tiene un conjunto de datos muy amplio, **no se pueden usar todas las instancias disponibles** en el procesamiento. En este caso una buena solución sería **hacer una selección de las**

instancias que mejor caractericen el sistema, pero para esto tendríamos que conocer muy bien la naturaleza de los datos y esto generalmente **no es algo trivial**.

La única solución viable y de fácil uso consiste en romper esta limitación que nos impone la computación *mono-máquina* **pasando a una *computación distribuida*** que nos permita procesar una cantidad ilimitada de instancias.

Una solución distribuida permite **usar todos los datos disponibles**, evitando así tener que realizar un filtrado que implique la necesidad de conocer la naturaleza del conjunto de datos.

Usando computación distribuida no resulta interesante realizar el filtrado de las instancias, de hecho aunque se realice de forma óptima, en el mejor de los casos no mejora las predicciones, solo obtendría una mejora en los tiempos de computación.

Esto es debido a que el algoritmo se auto-ajusta, se procesan todas las instancias y se eligen las mejores para realizar la predicción. Por muy bien que se haga el filtrado, esas mejores instancias seguirán siendo las mismas.

Haciendo un filtrado de los datos, en el peor de los casos, que se da con facilidad, se empeoran las predicciones. Ocurre cuando al realizar el filtrado se elimina alguna de esas mejores instancias al componer el subconjunto de los datos o que simplemente ocurra el hecho de que todas las instancias sean importantes y no entren en el subconjunto.

La variante del algoritmo con computación distribuida simplemente se limita a procesar todos los datos posibles con el fin de tener en cuenta más instancias a la hora de generar la predicciones, obteniendo de esta forma **una tabla de consulta que al ser más amplia hace más probable ajustar correctamente las predicciones** permitiendo encontrar una instancia igual o componer una solución con mayor número y mejores instancias similares que ajusten correctamente las predicciones.

Por un lado **se consiguen mejores predicciones** al poder procesar mayor cantidad datos y por otro, gracias a la computación distribuida podemos **paralelizar el procesamiento en varias máquinas**, dando la capacidad de obtener mejores tiempos. Los algoritmos de aprendizaje basado en instancias son altamente paralelizables, ya que el cálculo de distancias o la composición de la solución, que es lo que caracteriza este tipo de algoritmos, se pueden dividir en n subproblemas más pequeños. Si además contamos con que es un **modelo escalable**, basta con añadir más máquinas, tenemos una solución con unas capacidades ilimitadas.

Haciendo uso de una *computación distribuida en un algoritmo de aprendizaje basado en instancias* se puede componer una solución con un pasado infinitamente grande **sin ningún tipo de limitación**, eliminando así el principal problema que presentan este tipo de algoritmos **a costa de una solución más cara**.

Existen frameworks como **MapReduce** y **Apache Spark** que **facilitan la implementación de algoritmos distribuidos** y hacen posible de una forma sencilla usar este tipo de computación **siguiendo un paradigma**. Estas herramientas **son open-source**, no se tienen que pagar licencias, son totalmente accesibles y extensamente usados por la comunidad de desarrolladores. Hay muchos recursos y documentación disponible para abordar la tecnología.

1.4. Objetivos

Nuestra investigación gira en torno a **mejorar los algoritmos de aprendizaje basado en instancias usando una computación distribuida** para ajustar mejor las predicciones teniendo en cuenta un volumen de instancias impensable con una solución *mono-máquina*.

Dentro de los algoritmos de aprendizaje basado en instancias **nos hemos centrado en el algoritmo de regresión k-NN** con el fin de realizar predicciones **sobre series temporales**.

Una **serie temporal es un conjunto de observaciones que suceden en un intervalo de tiempo determinado**, puede tener una frecuencia temporal fija o variable y están ordenadas de forma cronológica. El análisis de estas series es interesante en campos como la climatología, las finanzas o la medicina. Se puede extraer información muy valiosa de ellas, realizar predicciones o estudiar futuras tendencias.

El algoritmo de regresión k-NN es un algoritmo simple y polivalente que nos permite realizar predicciones sobre series temporales. A grandes rasgos su funcionamiento consiste en estimar el valor de entrada usando su similitud respecto a un número (n) de observaciones previas, escogiendo una cantidad determinada (k) de mejores observaciones para usar los valores futuros de estas k mejores instancias para componer la solución.

Todos los algoritmos de aprendizaje basado en instancias tiene un grave inconveniente, los recursos necesarios para elaborar las predicciones crece de forma alarmante según va incrementando el volumen de datos. La cantidad de datos que puede procesar el algoritmo está limitada por los recursos de una máquina. Para solventar esta situación vamos a implementar el algoritmo usando computación distribuida que permita escalar los recursos disponibles usando varias máquinas, esto permite solventar sus limitaciones y hacerlo una opción más interesante.

Realizar la adaptación del algoritmo usando una programación distribuida es posible y garantiza todas las ventajas de que ofrece la computación clúster, debido a que el algoritmo es muy paralelizable. La gran mayoría de los cálculos que realiza, como calcular distancias, generar la matriz de distancias o componer la solución, se pueden realizar de forma parcial en subconjuntos de instancias más pequeñas y es esta capacidad de dividir el problema en n subproblemas más pequeños la que garantiza la eficacia de un algoritmo distribuido.

Nuestro trabajo culminará con el desarrollo de un algoritmo distribuido de regresión k-NN para series temporales en un entorno *BigData* que sirva como prueba tangible de que los algoritmos de aprendizaje basado en instancias, que son algoritmos que necesitan de un **histórico** para realizar las **predicciones**, pueden ser mejorados dejando a un lado la **programación tradicional y usando la programación distribuida**. Para poder tener en cuenta más cantidad de datos a la hora de realizar las predicciones. Además, se pueden conseguir mejores tiempos aprovechando la ilimitada escalabilidad horizontal.

Para la elaboración del proyecto nos hemos marcado los siguientes hitos:

1. Implementar el **algoritmo de regresión k-NN** para series temporales usando el motor de procesamiento distribuido Apache Spark. Entre los motivos de su elección destacamos:

    - Es la mejor opción para implementar algoritmos de aprendizaje automático, permite

cachear en memoria datos y es más cómodo para implementar procedimientos iterativos.

- Proporciona un entorno dinámico y ágil para el desarrollo, usando spark-shell se puede implementar funciones y verificarlas forma dinámica.
- Tiene una gran variedad de documentación y recursos disponibles.
- Oficialmente soporta los lenguajes de programación Scala, Java, Python y R, para este último teníamos una implementación de partida del algoritmo, se esperaba reutilizar partes del desarrollo pero finalmente se descarto debido a que no soportaba toda la funcionalidad de Spark, la API de R está limitada y no permite trabajar a grano fino con los datos. Hemos optado por un desarrollo integro en Python.

2. Un **estudio del impacto de diferentes configuraciones y arquitecturas** de computación distribuida. Este tipo de computación hace uso de varias máquinas, que implementan una configuración determinada, esto hace que cobre vital importancia la arquitectura y la jerarquía de los componentes. Gran parte del éxito de la solución implementada depende de la configuración y la arquitectura, estas siempre son objeto de optimización y tienen el mismo impacto que la implementación en sí del algoritmo.

3. Realizar una **comparativa con una implementación *mono-máquina*** del algoritmo. Se ha usado el trabajo realizado por los compañeros en su TFG: *"Librería para la predicción usando el k-NN: paralelización y visualización de resultados"* [1]. Nuestra comparativa se ha **centrado en las mejoras que ofrece procesar más cantidad de instancias en las predicciones**, las comparativas con diferentes parámetros de configuración del algoritmo ya fueron objeto de análisis en el trabajo de los compañeros.

Arquitectura, tecnologías y recursos empleados en el desarrollo son expuestos para abordar los conocimientos necesarios para la comprensión del trabajo realizado.

Capítulo 2

Contexto tecnológico

El funcionamiento y la optimización de un **algoritmo distribuido** es muy dependiente de las propiedades configurativas de cada componente del entorno que intervienen en el procesamiento distribuido, es de vital importancia conocer la tecnología, debemos adentrarnos en la **arquitectura y el funcionamiento de los componentes** para hacer un uso correcto de estas herramientas.

En este apartado haremos un desglose de las tecnologías empleadas tanto para el despliegue como para la **implementación de la solución**.

2.1. Apache Hadoop

Es un proyecto *open-source* de *Apache foundation* centrado en el desarrollo de componentes de **computación distribuida, escalable y tolerantes a fallos** [1] [5].

Estos componentes proporcionan una capa de almacenamiento y procesamiento datos con una gestión de usuarios, privilegios, control de accesos y una seguridad.

Los componentes que engloban el ecosistema de *Apache Hadoop* son las herramientas más utilizadas para la computación y el almacenamiento distribuido.

Su éxito en gran medida es debido a su gran capacidad de procesamiento y almacenamiento, así como bajo coste de implantación, mantenimiento y escalado.

Un **clúster Hadoop** es un conjunto de máquinas que implementan uno o varios componentes de Hadoop. Generalmente siguen el modelo maestro-trabajadores.

Cada componente de *Hadoop* tiene un servicio que se encarga de orquestar las tareas y otros servicios que están en la misma máquina o en otras que se encargan de realizar el trabajo en sí.

Cabe destacar la adaptabilidad, flexibilidad y la capacidad de evolución de los entornos *Hadoop*.

El *clúster* se adapta correctamente a cualquier necesidad del negocio, la funcionalidad del entorno se amplía añadiendo más componentes a nuestro *clúster* consiguiendo un **entorno correctamente adaptado a las necesidades del negocio** permitiendo aprovechar todos los recursos del *clúster*. Posteriormente si las necesidades se amplían se pueden añadir de una forma rápida y sencilla más componentes al clúster.

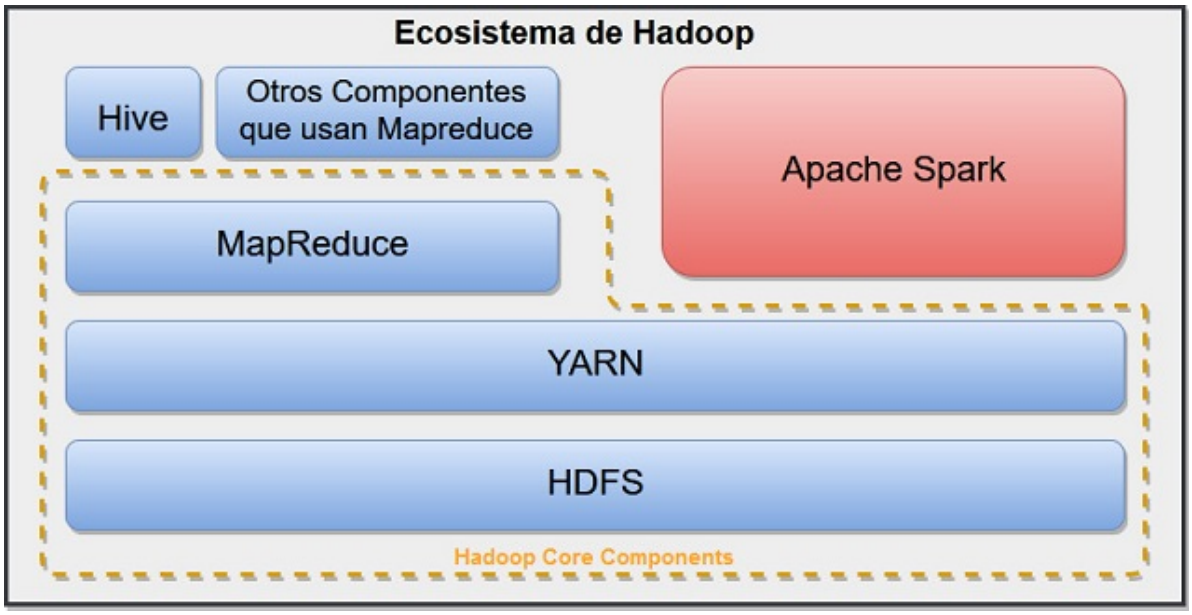

Figura 2.1: Componentes del ecosistema de Hadoop con su jerarquía de dependencias.

2.1.1. HDFS

Es un componente que engloba la **capa de almacenamiento distribuido**, perteneciente a *Hadoop Common*, que es la base de *Hadoop* y se encarga de proveer un sistema de ficheros distribuido, escalable, seguro y tolerante a fallos [8].

Hay que tener en cuenta que no es un sistema de ficheros con propósito general, el conjunto de operaciones permitidas está limitado, no se pueden modificar los datos y esto es debido a que el sistema está pensado para hacer lecturas. Entre muchas de sus virtudes caben destacar:

- Mejora el rendimiento de los motores de procesamiento distribuido del ecosistema de *Hadoop* como *MapReduce* o *Spark*, esto es debido a que el procesamiento se lleva a la máquina donde se encuentra físicamente el dato. Este fenómeno se le conoce como **data locality**, consiste en elegir nodos que tiene el dato físicamente dentro del *clúster* para ser los que realicen el procesamiento, con ello se evitan latencias y transferencias de red.

- Tiene jerarquía de directorios como el sistema tradicional de *Linux*, con su gestión de permisos y usuarios por directorio.

- Permite el almacenamiento de enormes cantidades de datos, con un almacenamiento fácilmente escalable y barato, basta con añadir más máquinas o ampliar la memoria de las ya existentes.

- Es un sistema de ficheros fiable gracias a la **replicación de los datos** que evitan su pérdida y mejora el rendimiento de las lecturas.

2.1.2. YARN

Es un gestor de recursos del clúster para motores de procesamiento distribuidos, puede ser usado además de por MapReduce por diferentes motores de procesamiento como Spark u otros componentes del ecosistema de Hadoop [8].

Estos componentes a través de una API solicitan los recursos que necesitan del clúster para llevar a cabo sus trabajos. Además monitoriza el uso de los recursos y los trabajos que están siendo ejecutados dentro del clúster.

2.1.3. MapReduce

Aunque finalmente no se ha elegido como motor de procesamiento distribuido para la implementación del algoritmo, es interesante darle una mención. Los motivos de su descarte para la implementación del algoritmo se exponen en las secciones 2.2.4 y 2.3.
MapReduce surge como solución al procesamiento de grandes volúmenes de datos que no podían ser procesados con un modelo mono-máquina.
Es un framework para el procesamiento de algoritmos paralelizables y distribuidos.
Pertenece a Hadoop Commons. Usa el paradigma de programación MapReduce.

Este paradigma está inspirado por la estrategia de programación Divide y Vencerás, que consiste en dividir el problema original en n subproblemas más sencillos para resolverlos de forma individual y luego combinar todas las soluciones [8]. MapReduce consta de tres fases:

1. **Map**: se paralelizan y se distribuyen los datos en formato clave-valor en cada máquina y se les aplica una función.

2. **Shuffle & Short**: se aplica a la salida de la fase Map un algoritmo de shuffle and sort que se encarga de agrupar los datos por clave para que sirvan como entrada a la fase Reduce.

3. **Reduce**: Es el último paso, en la fase Reduce, por clave se combinan las salidas de los Map aplicando una función determinada.

Un trabajo MapReduce solo requiere implementar la parte Map, la parte Reduce no es obligatoria, debido a que en muchos contextos no es necesario hacer una reducción de los datos. Por ejemplo, si queremos implementar un algoritmo que haga el volcado de una base de datos al sistema de ficheros HDFS, solo tendríamos que implementar la parte Map que se encargue de sacar los datos y los mande al sistema de ficheros.

MapReduce permite cualquier entrada de datos, sea de componentes de Hadoop como HDFS o componentes externos como bases de datos.

2.2. Apache Spark

Spark es un motor de procesamiento distribuido con un conjunto de librerías para el procesamiento paralelo en un clúster, es open-source, escalable y tolerante a fallos [7].

Surge debido a la necesidad de implementar un motor de procesamiento distribuido que fuera eficiente para tareas en las que se hagan varias iteraciones sobre un conjunto de datos, estas tareas

usando MapReduce son totalmente ineficientes puesto que para cada iteración hay que lanzar otra tarea MapReduce, véase *figura* 2.5.

Tiene su propio motor de procesamiento distribuido, no usa MapReduce como hacen otros elementos del ecosistema de Hadoop como Hive o Pig. Este motor tiene la capacidad de mantener en memoria gran cantidad de datos. Algunos de los motivos para elegir Spark son:

- Trabajar con *Spark* resulta atractivo puesto que se adapta a un abanico bastante amplio de usuarios como programadores o analistas.
- Aparte de Java soporta más variedad de lenguajes que MapReduce, como Python, Scala o R. Este último lenguaje presenta ciertos problemas debido a que no soporta la totalidad de la funcionalidad de Spark.
- Permite trabajar con los datos al estilo de tablas de SQL gracias a las librerías de SparkSQL.
- Permite el procesamiento de datos en near-real-time usando las librerías de SparkStreaming, este tipo de procesamiento consiste en : dado un dato en Streaming realizar su procesamiento en un instante de tiempo inferior a un minuto.
- Incluye librerías de machine learning como MLlib o bibliotecas de procesamiento gráfico como GraphX.
- Soporta la mayoría de sistemas de almacenamiento distribuidos como HDFS, Amazon S3, Azure Storage.
- Cuenta con una gran comunidad de desarrolladores que mantienen y mejoran el proyecto con nuevas librerías y mejoras.

2.2.1. Arquitectura interna

Las aplicaciones de *Spark* están formadas por un proceso Driver (es el que gestiona todo el trabajo de Spark) y un conjunto de ejecutores (son un conjunto de procesos que realizan el procesamiento distribuido y son gestionados por el Driver). El Driver ejecuta la función principal del programa dentro de un nodo y se encarga de hacer un análisis de los trabajos, distribuir y programar la ejecución del trabajo Spark en los ejecutores. Spark usa un gestor de recursos (YARN, Mesos, Standalone ...) para aprovisionar los ejecutores necesarios para realizar el procesamiento de los datos.

Apache Spark soporta varios lenguajes de programación pero su forma de trabajar con ellos difiere en función de si puede ser ejecutado en una JVM (Java Virtual Machine) o no. Java y Scala pueden ser ejecutados directamente pero el resto de los lenguajes soportados (Python y R) se traducen a un código que puede ser interpretado por las JVM de los ejecutores.

Modos de despliegue

Las aplicaciones Spark se distribuir a lo largo del clúster apoyándose en gestores de recursos. algunos de los gestores de recursos son:

- Standalone mode: Es un gestor de recursos creado específicamente para Apache Spark, es una buena opción si en el clúster solo se ejecutan tareas de Spark debido a que es sencillo y rápido, pero si en el clúster tenemos una administración y una gestión concreta de los trabajos con prioridades o límites de recursos y además se ejecutan otros procesos distribuidos que no sean de Spark, no sería una buena elección.
- Apache Hadoop YARN: En apartados anteriores se habló de este gestor de recursos, las ventajas que presenta es que está integrado por defecto en un clúster Hadoop y su complejidad permite que coexistan Spark con otros motores de procesamiento.

Modos de ejecución

Las aplicaciones Spark se pueden ejecutar de 3 formas:

- Modo Local: Es el modo de ejecución en la propia máquina local, el paralelismo se realiza mediante hilos, es el modo indicado para hacer pruebas o desarrollos de una forma rápida.
- Modo Cliente: En este modo la aplicación despliega en el propio cliente el Driver y es el cliente el que mantiene el proceso, si se cae la conexión del cliente con el clúster la aplicación se cierra. Esta es la desventaja de este modo pero este el único modo distribuido que soporta Spark Shell, esta herramienta es muy útil para realzar el desarrollo y las pruebas ya que permite hacer sentencias interactivas de una forma muy rápida.
- Modo Clúster: En este modo el cliente despliega un ejecutable dependiente del lenguaje de programación. El gestor de recursos lanza en un nodo trabajador el proceso Driver de Spark y es el gestor el que tiene que mantener el proceso. En este modo de ejecución no necesitamos que esté activo el cliente durante todo el proceso.

2.2.2. Flujo de una aplicación Spark

El código que lanzamos en Spark lanza una aplicación de Spark, la aplicación esta por compuesta por uno o varios Jobs que generalmente se ejecutan uno detrás de otro. Estos Jobs están compuestos por Stages, que a su vez se dividen en Tasks. Spark automáticamente analiza Stages y Tasks con el fin de realizar el procesamiento de una forma óptima.

Para entender el flujo de una aplicación *Spark* es importante explicar los siguientes elementos de *Spark*:

Jobs

Un trabajo en Spark es un conjunto de Stages, habrá tantos Stages como shuffles (reparticiones) se hagan con los datos.

SparkSession

Es el punto de entrada de la aplicación desde Spark 2.0, para lanzar un proceso *Spark* es necesario obtener la instancia desde el *Driver* de este objeto. Desde este objeto se hacen las lecturas

del conjunto de datos de entrada en el formato que se prefiera RDD, DataFrame ...

Particiones

Cuando hacemos una lectura de los datos con en el *Driver* con el objeto *SparkSession* para fines de paralelización distribuida a lo largo del clúster se realizan particiones sobre el conjunto de los datos de entrada. Las particiones se distribuyen en los distintos ejecutores que se lanzan con la aplicación. Cada partición se procesa en un Task del ejecutor, un Task no es más que un procesamiento paralelo que ocurre en un ejecutor. Es conveniente tener al menos las mismas particiones que ejecutores. Si los datos son muy grandes es conveniente añadir más particiones debido a que con la configuración por defecto el shuffle no puede superar más de 2 Gb. Las particiones son objeto de optimización en los trabajos de Spark.

Transformaciones

En Spark los objetos son inmutables, es decir que no se pueden modificar, en vez de esto se realizan una lista de modificaciones para posteriormente crear otro objeto derivado con los cambios, a eso se le llama transformación.

La generación de estos nuevos objetos llevan un modo de ejecución *lazy*, se siguen añadiendo modificaciones a la lista de cambios (plan de ejecución) sin generar el nuevo objeto hasta que no se necesita forzosamente. En jerga de Spark, no se genera hasta que no ocurre una Action, mientras que no ocurre todas las transformaciones se realizan en el mismo Stage. Las transformaciones van generando un plan de ejecución que no es ejecutado hasta que ocurre una Action. Las transformaciones dan lugar a dos tipos de dependencias:

- Narrow dependency : esta dependencia surge cuando las transformaciones se realizan sobre las particiones 1 a 1 y no necesitan repartición de los datos, se van añadiendo a él plan de ejecución.

- Wide dependency : Exigen de una repartición de los datos, desemboca a realizar una Action que despliega todo el plan de ejecución dando como resultado un nuevo Stage.

Spark con su plan de ejecución optimiza todas las transformaciones, pero no tiene en cuenta los datos que se encuentran entre Stages. En caso de reutilización es conveniente hacer un cacheo.

Actions

Es el desencadenante que fuerza a el lanzamiento del plan de ejecución con las transformaciones que están previstas sobre los datos para un Stage. Se produce cuando hay una *Wide dependency* en el plan de ejecución que fuerza a generar los datos. Operaciones como count, collects o escribir el resultado de las transformaciones generan una Action.
Las Actions siempre devuelven resultados.

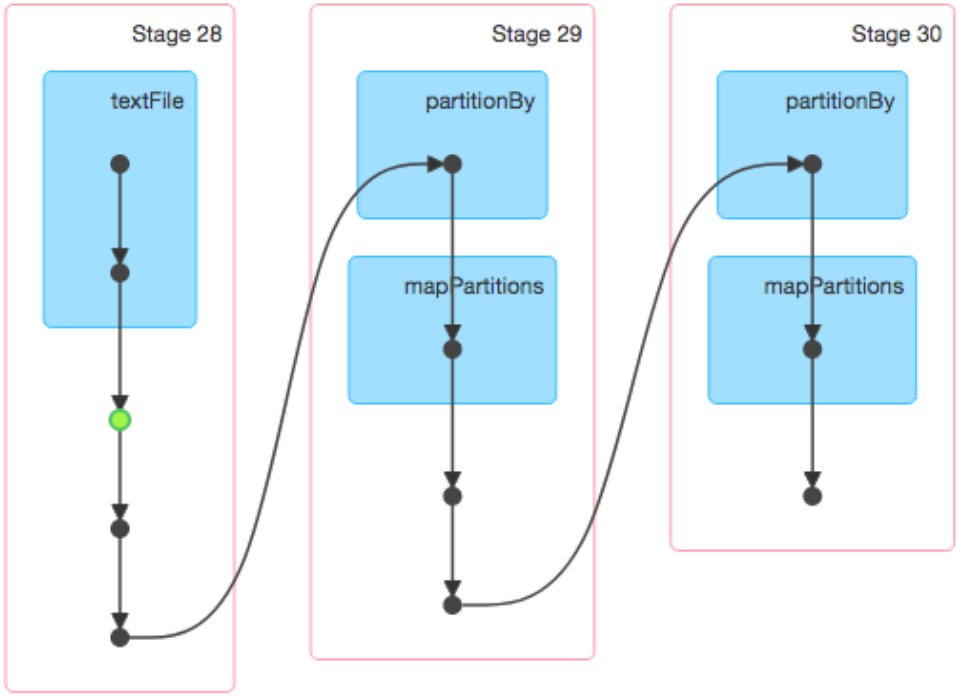

Figura 2.2: Captura de una parte del flujo del algoritmo.

Como podemos observar en la *figura* 2.2 cada vez que se realiza un partitionBy (es una Action) este da lugar a un nuevo Stage, esto se debe a que la función partitionBy necesita realizar una repartición de los datos, esta función genera una Wide dependency. Esta figura es una visualización de un DAG generado en un trabajo de Spark, representa el flujo previsto por el planificador de tareas, los cuadros azules representan las funciones que se ejecutan y las líneas representan las transformaciones de los datos.

Usar esta información que proporciona la visualización del DAG es muy útil para optimizar el código ya que nos dan información sobre funciones que se realizan en un mismo Stage o Stages que dependen unos de otros, si somos capaces de interpretar correctamente la visualización se pueden evitar cuellos de botella innecesarios y se puede optimizar el código haciendo un uso adecuado de las funciones, todas las funciones que se realizan en un mismo Stage son optimizadas automáticamente pero las que están entre Stages no se optimizan, es tarea del desarrollador visualizar y optimizar los procesos que ocurren entre Stages.

Stages

Son un conjunto de tareas que se puede o transformaciones que se puede llevar a cabo sin realizar una repartición de los datos, esto ocurre mientras que no se tenga un Wide dependency que fuerce a realizar un shuffle de los datos.
Acumulando estas tareas se optimizan los cálculos evitando reprocesamientos y reparticiones innecesarias.

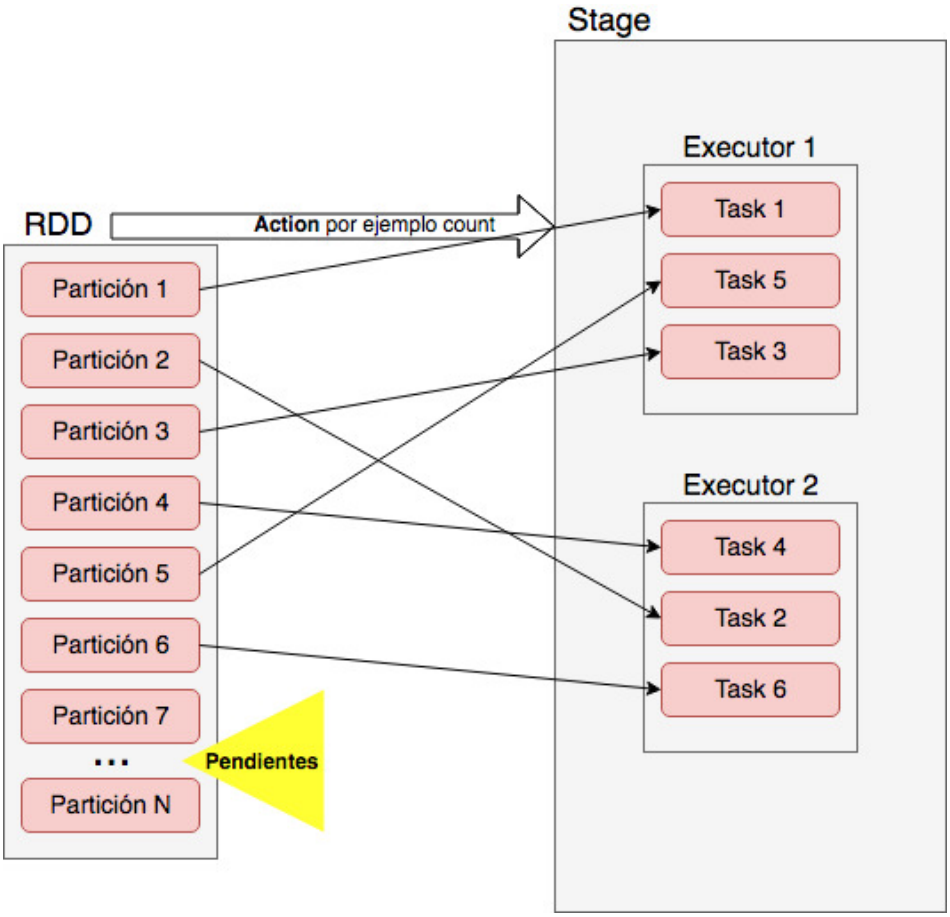

Figura 2.3: Secuencia con la generación de un Stage.

En la *figura* 2.3 podemos observar que se generan tantas Tasks como particiones tenga el RDD, estas serán N. Todas estas Tasks (1 ... N) se van procesando según van quedando huecos libres en los ejecutores en un mismo Stage. Como podemos observar en la figura cada ejecutor es capaz de procesar tres tareas de forma paralela, teniendo dos ejecutores como los de la imagen, podemos paralelizar hasta 6 tareas, cada tarea procesa una partición de los datos. Al terminar procesarse estas Task que se visualizan en la imagen (Tasks 1-6), irían entrando las siguientes Tasks 7-N, sin seguir ningún orden pero manteniendo el procesamiento máximo de seis Tasks de forma paralela. El factor de paralelismo (cantidad de tareas paralelas) de cada ejecutor depende del número v-Cores que tenga asignados, en este ejemplo como máximo cada ejecutor puede procesar tres Task por lo tanto podemos deducir que cada ejecutor cuenta con 3 v-Cores. En el ejemplo de la *figura* 2.4 se puede observar mejor esta relación de v-Cores y Tasks.

Task

Las tareas son operaciones entre bloques de datos, estas transformaciones ocurren en un mismo ejecutor que procesa una partición de los datos, cuantas más particiones se tengan más paralelismo habrá y mejores tiempos obtendremos.

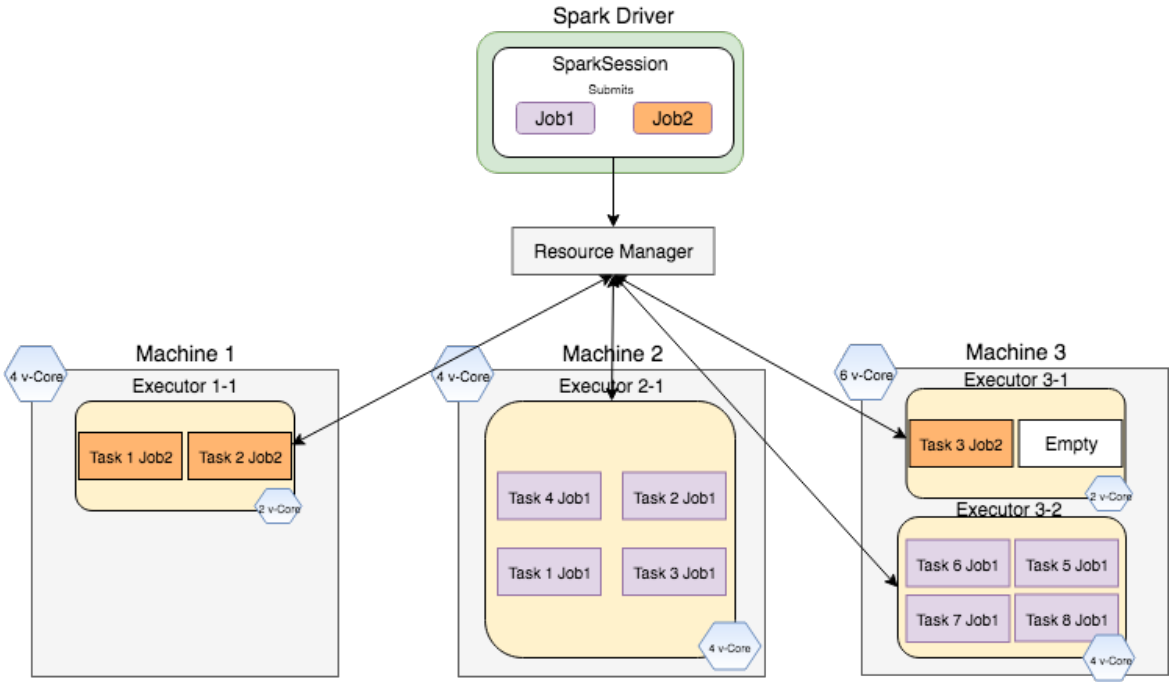

Figura 2.4: Distribución de las Tasks que confluyen de forma concurrente en un clúster

La figura 2.4 representa dos Jobs que están haciendo uso del clúster de forma concurrente. El clúster esta formado por tres máquinas, dos con 4v-Cores y una con 6v-Cores. Como podemos deducir de la imagen, los Jobs usan configuraciones diferentes, para Job1 se están usando ejecutores de 4v-Cores en cambio para Job2 de 2v-Cores. Se puede observar que la cantidad máxima de Tasks que se pueden ejecutar en un Job de forma paralela es su número de ejecutores multiplicado por el número de v-Cores por ejecutor. Dentro de los ejecutores se lanza una JVM con tantos procesos paralelos como vCores tenga la configuración de ejecutor. Es importante cuadrar bien la configuración de memoria y vCores por ejecutor para aprovechar mejor los recursos del clúster. Podemos observar en la imagen que si el Job2 hubiese usado ejecutores de 2vCores en vez de 4vCores, hubiera cambiado la distribución de ejecutores teniendo en la máquinas 2 y 3 dos ejecutores de 2v cores en vez de uno de 4v-cores, permitiendo desplegar un ejecutor adicional en la máquina 1 (solo tenia 2v-Cores disponibles,ahora si hay recursos para la nueva configuración de ejecutor), permitiendo procesar hasta 10 Tasks de forma paralela en vez de 8, aprovechando mejor los recursos del clúster. Por otro lado tenemos el Job2 que no ha tenido configurado correctamente el número de particiones de los datos, esto se observa en el ejecutor 3-1, solo procesa una tarea a pesar de que su configuración le hubiera permitido procesar dos. Esto no permite aprovechar correctamente los recursos del clúster incrementando los tiempos de procesamiento, es necesario ajustar correctamente las particiones en función de la configuración de los ejecutores.

Usar una configuración con muchos vCores fuerza a tener menos ejecutores que tienen gran cantidad de memoria. Una configuración con muchos vCores nos ofrece las ventajas del paralelismo en una JVM como el uso de memoria compartida pero si contamos con demasiada memoria por ejecutor, el Garbage Collector ralentiza demasiado el proceso al gestionar una gran cantidad

de memoria.
Usar una configuración con muchos ejecutores fuerza a inutilizar gran parte de la memoria empleándolas en las JVM que tiene que gestionar cada ejecutor en vez de usarlas para el procesamiento, con esta configuración es más sencillo hacer uso de todos los recursos del clúster pero no consigue un aprovechamiento óptimo.
Hay que buscar un equilibrio entre ambas para conseguir una buena gestión de los recursos y un buen aprovechamiento, evitando que gestione demasiada memoria por JVM e intentando tener las menos JVM posibles para evitar replicar variables de Broadcast en cada JVM y gastar memoria en la gestión del proceso.

En nuestro caso particular utilizamos variables de Broadcast bastante grandes y en ellas guardamos toda la parte de las instancias temporales que integran la parte de entrenamiento (véase *figura* 3.1), esta variable de Broadcast se distribuye a cada ejecutor. La configuración más adecuada a utilizar en este algoritmo es tener menos ejecutores con más vCores usando un número de particiones que garantice el aprovechamiento del clúster, una configuración correcta sería : $n^{o}{}_{particiones} = n^{o}{}_{ejecutores} * vCores_{ejecutor}$. Menos ejecutores implica menos réplicas de la variable de Broadcast en el clúster (una réplica por ejecutor). Este es un claro ejemplo que refuerza la idea que se presentó en la introducción, los algoritmos distribuidos dependen en gran medida de su configuración.

2.2.3. Tipos de datos y objetos

RDD

Es la abstracción básica que tiene Spark de un conjunto de datos distribuido en varias máquinas, es tolerante a fallos y se puede cachear para optimizar las aplicaciones.

DataFrame

Un DataFrame es un tipo de datos que representan la estructura de una tabla que tiene filas y columnas, pertenecen a la librería SparkSQL. Tiene su esquema con el tipo de dato de cada una de las columnas. Simplifica las operaciones representando los RDD como si fueran una tabla.

Accumulators

Los acumuladores son variables compartidas que solo tienen definida una operación de add, como contadores.

Broadcast Variables

Una variable de Broadcast es una variable que se serializa y que se envía a cada ejecutor para que las tareas puedan usarla en el caso de necesitarlas, son similares al Distributed Cache de MapReduce. Generalmente se almacenan en memoria, pero en el caso de que no hubiera memoria se escribirían en disco.

2.2.4. Ventajas de Spark respecto a MapReduce

A la hora de elegir un framework de procesamiento distribuido para el algoritmo k-NN hemos tenido en cuenta las siguientes ventajas que nos ofrece Apache Spark respecto a MapReduce:

- Para trabajos iterativos sobre un conjunto de datos en los que se reutiliza información es bastante más rápido que MapReduce.

- Gracias al planificador de tareas DAG, se puede trazar y monitorizar de una forma precisa los trabajos que realiza por aplicación. Esto es muy útil a la hora de optimizar el código.

- Es más sencillo que MapReduce para el programador, además de Java soporta varios lenguajes de programación como Python, Scala o R.

- Permite trabajar de forma sencilla con datos en memoria, los cacheos de datos que se reutilizan mejoran los tiempos de procesamiento del algoritmo.

- Desarrollar con Spark es muy rápido ya que cuenta con un modo interactivo que a través de una shell se pueden desarrollar y testear funciones de forma rápida y sencilla.

Otras características que ofrece Apache Spark y que no han sido usadas en el proyecto son:

- Tener librerías que facilitan la programación como SparkSQL que permite hacer consultas al estilo SQL.

- Contar con librerías como MLlib para tipos de datos y algoritmos usados en machine learning.

Es importante mencionar que en un intento de reutilizar partes del algoritmo ya implementadas en R [1], pudimos comprobar que la API de Spark en R esta limitada a el uso de SparkSQL y a trabajar con funciones de tabla, no permite trabajar con los datos a bajo nivel, esto nos hizo descartar la opción de usar R y reutilizar parte de la solución implementada por nuestros compañeros.

Para nuestro caso en particular, que realizamos procesamientos iterativos sobre un conjunto de datos (véase la función de la *figura* 4.9) Spark ha sido la mejor opción pero debemos tener en cuenta que en otros contextos en los que se den procesamientos lineales simples de grandes cantidades de datos lo mejor es usar MapReduce.

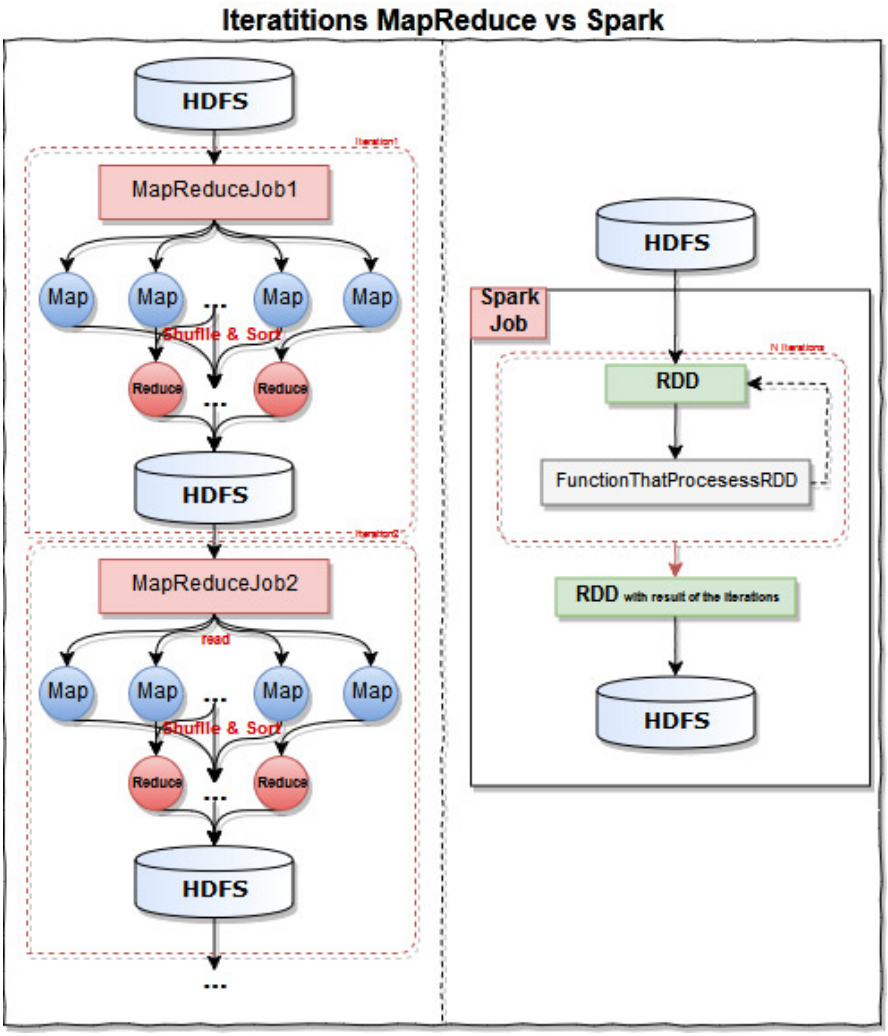

Figura 2.5: Iteraciones MapReduce vs Spark.

Como se puede observar en la *figura* 2.5, realizar procesos iterativos en MapReduce no es sencillo ni rápido, requiere para cada iteración definir un nuevo trabajo MapReduce que lea del sistema de ficheros y escriba los resultados parciales en el sistema de ficheros para posteriormente en la siguiente iteración volver a definir otro trabajo MapReduce y volver a leer.

Tener que volver a definir trabajos de MapReduce lastra enormemente los tiempos de ejecución, para cada trabajo hay un negociado de los recursos entre MapReduce y el gestor de recursos utilizado que impone un tiempo muerto se que repite con cada iteración. En cambio con Apache Spark no es necesario ni volver a definir nuevos trabajos ni guardar en memoria procesamientos parciales, este trabaja con memoria.

Teniendo en cuenta todo lo anterior podemos concluir afirmando que nuestra solución en Spark es mejor que la que se hubiera podido obtener usando MapReduce.

2.3. Presentación de la tecnología empleada

El proyecto requería el despliegue de un entorno *open-source* que permitiera realizar una computación distribuida con una escalabilidad fácil y económica, que contase con abundante documentación accesible y el respaldo de una comunidad de desarrolladores.

Se ha apostado el proyecto *open-source Apache Hadoop* dando uso a las siguientes tecnologías que conforman el ecosistema de Hadoop: *Hadoop Core* y *Apache Spark*.

Hadoop Core nos proporciona un **gestor de recursos** del *clúster* (*YARN*), es el gestor de recursos que hemos elegido para los trabajos en Spark, debido a que se realizaron pruebas con el gestor de recursos que trae integrado Spark por defecto (Spark Standalone) y se obtuvieron peores resultados en tiempos de ejecución para las mismas tareas que con YARN.

Además, Hadoop Core proporciona un **sistema de ficheros distribuido** (*HDFS*). Este sistema de ficheros se ha utilizado como entrada de datos para los trabajos Spark, aquí almacenamos las series temporales. Es interesante tomar la entrada de los datos desde HDFS gracias a que su integración con Spark, la replicación de los datos y la división en bloques optimiza el procesamiento distribuido de las series temporales.

El motor de procesamiento distribuido incluido en *Hadoop Core* (*MapReducev2*) no ha sido la solución utilizada por los siguientes motivos:

- A nivel de desarrollo es complejo y tedioso, la API no es intuitiva y no acepta variedad de lenguajes de programación, solo se limita a Java.
- No es eficiente en procesos iterativos. En el algoritmo k-NN hay tareas como la optimización de los parámetros k y d que son procesos iterativos.
- No trabaja con memoria, por lo tanto no se puede realizar el cacheo de datos previamente calculados, si hay reprocesamiento se volverían a recalcular los datos de nuevo. En el algoritmo k-NN, en puntos como la elección del mejor parámetro k se reutiliza la matriz de distancias, sino podemos cachearla se tiene que volver a recalcular y este procesamiento es el más costoso del algoritmo.
- No permite el procesamiento de los datos en micro-batch o near real-time. Este tipo de procesamiento se realiza en un intervalo de tiempo pequeño, menos de 1 minuto y permite realizar procesamientos en Streaming según llegan los datos. Usando el algoritmo k-NN para predicción de series temporales no tiene sentido este procesamiento, pero en otro escenario usando clasificación puede llegar a ser interesante hacer predicciones según llegue la nueva instancia a predecir.

El motor de procesamiento elegido ha sido *Apache Spark* puesto que a nivel de desarrollo es amigable y soporta lenguajes de programación populares como *Java*, *Python*, *R* y *Scala*. Es un motor más rápido debido a que trabaja con los datos en memoria además de en disco y no presenta los problemas anteriores indicados con **MapReduce**.

La elección de estas tecnologías ha estado motivada nuestro conocimiento previo en estas así como por su enorme adopción y aceptación de las mismas en la comunidad de desarrolladores.

2.4. Recursos utilizados en el desarrollo

En este apartado se expondrán todos los recursos con los que hemos contado a la hora de elaborar este proyecto, tanto los usados en el desarrollo como en las pruebas.

2.4.1. Ordenador portátil

El uso de este computador se ha centrado en desarrollar la implementación del algoritmo distribuido y en realizar las pruebas comparativas entre la solución mono-máquina implementada por los compañeros en R y la solución distribuida de Spark ejecutada en el clúster. Su configuración es la siguiente:

- **Sistema Operativo** : Windows 10 - 64 bits.
- **Procesador** : Intel(R) Core(R) CPU I7-8550U @ 1.99GHz.
- **Memoria ram** : 8 Gb.

2.4.2. Máquinas

En un desarrollo centrado en la creación de un algoritmo de computación distribuida es imprescindible contar con varias máquinas interconectadas que permitan procesamiento paralelo formando un clúster.
Hemos contado con 5 instancias virtuales con la siguiente configuración por instancia:

- **Sistema Operativo** : CentOS 6.
- **Procesador** : 6 x Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz.
- **Memoria RAM** : 16 Gb.

2.4.3. Clúster Hadoop

La configuración del clúster Hadoop incluye los siguientes servicios con la siguiente arquitectura:

Spark

Binarios de Spark: nodo1.

YARN

- **ResourceManager** : nodo1.
- **NodeManager** : [nodo2-nodo4].

Los recursos que se asignan por nodo para ser utilizados por los trabajos distribuidos en los nodos trabajadores o esclavos son 14 Gigas de memoria RAM y 6 v-cpus, se reservan 2 Gigas de memoria para los servicios activos y el sistema operativo. En total contamos con 58 Gb y 24 v-cores para este fin.

HDFS

- **NameNode** : nodo1.
- **DataNode** : [nodo2-nodo4].

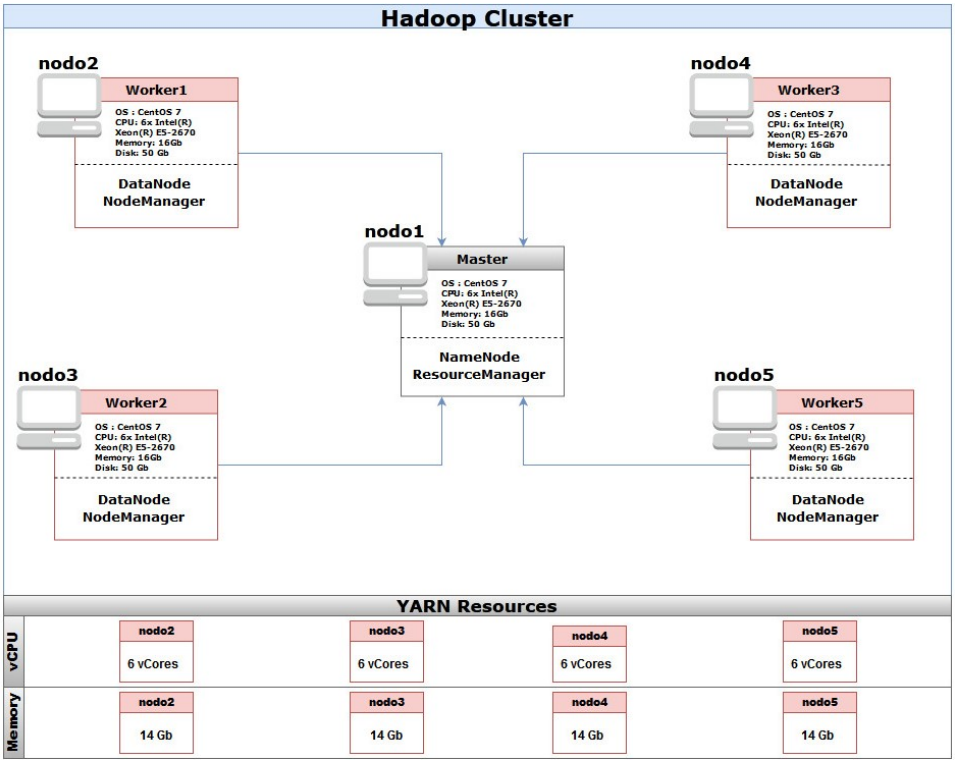

Figura 2.6: Diagrama con los recursos y servicios del Clúster de Hadoop

2.4.4. Amazon EMR

Con el fin de lanzar pruebas en un entorno más productivo, con máquinas mejores que las de nuestro clúster de Hadoop, hemos usado Amazon Elastic Mapreduce (EMR) para ejecutar los trabajos Spark.

Este componente de Amazon Web Services (AWS), aprovisiona de forma sencilla, rápida y escalable un clúster Hadoop usando instancias de EC2 (máquinas virtuales de AWS).

Al usar una nueva cuenta de AWS no hemos podido desplegar un clúster con muchos recursos debido a que de partida todas las cuentas están sujetas a limitaciones, posteriormente se puede solicitar un incremento de la cuota, con estas limitaciones como máximo se pueden lanzar 20 instancias de EC2, además hay un límite de instancias por tipo de máquina, estas limitaciones son dependientes de la región en la que se lancen las tareas de EMR.

El clúster más potente que se ha lanzado para ejecutar un trabajo Spark ha tenido la siguiente configuración:

- **máster** : 1x m4.large.
  - **CPU virtual** : 4.
  - **Memoria RAM** : 8Gb.
- **Core** : 10x m4.xlarge.
  - **CPU virtual** : 8.
  - **Memoria RAM** : 16Gb.

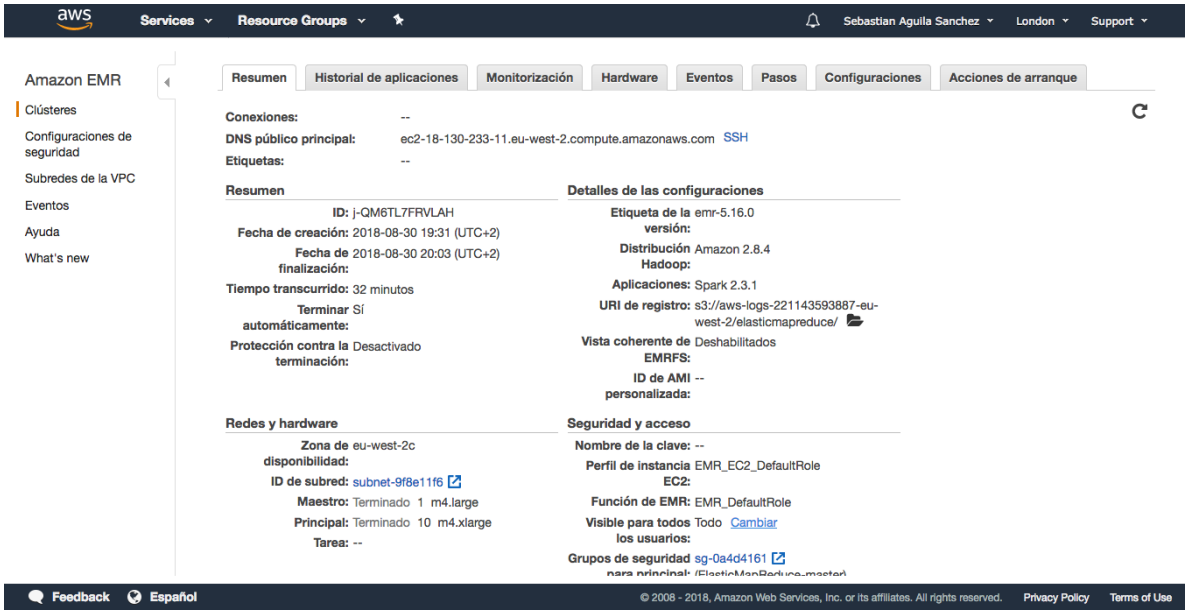

Figura 2.7: Ejemplo de un clúster de EMR desde la interfaz web de AWS.

Capítulo 3

Algoritmo *k*-Nearest Neighbours

El algoritmo ***k*-Nearest Neighbours (*k*-NN)** es un algoritmo de aprendizaje basado en instancias (ejemplos). A este tipo de aprendizaje se le conoce como ***Lazy Learning***, ya que, no busca un modelo para resolver el problema cuando se presentan nuevas instancias, sino que **utiliza las instancias ya dadas para entrenarse** y a través de dichas instancias ya dadas, trata con la nueva instancia que se presenta basándose en lo aprendido, para ello usa las k instancias vecinas más cercanas del entrenamiento para tratar nueva instancia que aparece. Este algoritmo se utiliza de dos maneras que son:

- **Clasificación**: Esta manera **clasifica las instancias de forma supervisada**, al presentarse nuevas instancias, se toma la clase más representada en las k instancias seleccionadas usando una **función de densidad** que estima lo probable que, una instancia pertenezca a una clase u otra, puede usarse por ejemplo una **votación ponderada por cercanía**.

- **Regresión**: Esta manera del algoritmo, todos las instancias disponibles y predice un valor numérico de una nueva instancia que se vaya a presentar, en función de hacer una **media de los *k* instancias vecinas almacenadas a la nueva instancia que se presenta**, la obtención de los k vecinos es igual a la de clasificación, calculado la distancia a la instancia que se presenta.

En nuestro caso usaremos instancias de **series temporales** que llamaremos **instantes temporales** usando el método de **regresión**.
El ***k*-NN en series temporales** consiste en localizar los k instantes del pasado de la serie temporal que más se parecen al instante actual para generar una predicción, a través del valor siguiente valor de las serie temporal a dichos k instantes vecinos del pasado, la caracterización de un instante temporal se hace en base a sus d últimos valores de instantes en la serie.
Este algoritmo es muy útil porque es sencillo de aplicar y versátil a la hora de querer predecir acontecimientos a futuro en datos económicos (cambios del PIB, evaluaciones de monedas virtuales...), efectos medio ambientales (temperatura,humedad...) o cualquier dato del cual se pueda evaluar los datos obtenidos del pasado de una serie temporal.

3.1. Descripción del Algoritmo

Como hemos comentado en anterioridad, este algoritmo *k-NN* predice sobre **una serie temporal** [3]. Para realizar las predicciones se realizan los siguientes pasos [2]:

1. Se preparan los datos, **agrupando *d* instantes temporales del pasado** para poder calcular la distancia entre instantes temporales. Cada instante temporal se representa por sus d últimos valores. De esa forma una serie temporal se transforma en una matriz de instantes temporales descritos cada uno por d valores (o variables).

2. Calculamos las distancias de cada todos los instantes temporales que queremos predecir contra todos sus instantes del pasado usando medidas de similitud, estas distancias serán almacenadas en lo que conoceremos como **matriz de distancias**.

3. Elegimos en la matriz de distancias **los *k* instantes del pasado con menor distancia** (*k* vecinos) a los instantes temporales que queremos predecir.

4. Una vez obtenemos los vecinos, usamos el valor de predicción de los *k* vecinos más cercanos y calculamos una **media ponderada** que dará como resultado la predicción al instante temporal que estamos evaluando.

Una vez completados estos pasos hay que entender cómo funcionan exactamente los cálculos para completar el algoritmo. Este usa exactamente dos que son:

- **Distancias o medidas de similitud**: Se utilizan para encontrar los **conjuntos más parecidos** (vecinos) al instante temporal que se está evaluando para poder predecir sobre los mismos revisando todo el pasado al instante evaluado.

- **Generación de la predicción**: Ya obtenidos los vecinos más cercanos, es decir, los *k* instantes en el pasado de la serie más parecidos al instante actual que estamos evaluando, utilizamos el siguiente valor de cada uno de los *k* instantes y realizamos una media ponderada por la cercanía para obtener una predicción calculada con la predicción de los *k* vecinos.

A continuación, hablaremos de estos cálculos que se han implementado con más en profundidad.

3.1.1. Parámetros de predicción

Cuando aplicamos el algoritmo *k*-NN han de fijarse los parámetros *k* y *d* a los cuales haremos referencia durante todo el capítulo el significado de cada parámetro es el siguiente:

- **Parámetro *k***: Este parámetro fija el **número de vecinos más cercanos al instante temporal que estemos evaluando**, estos *k* vecinos servirán para realizar la predicción de cada instante temporal en función de la **media** de los valores de predicción cada *k* vecino.

- **Parámetro *d***: Sirve para **preparar los datos para el algoritmo *k*-NN**, fijar cuantos el **número de retardos**, es decir, valores anteriores y consecutivos, que usamos para describir cada instante temporal.

3.1.2. Medida de similitud

Para conocer los elementos más cercanos al los conjuntos que estamos evaluando usamos una **medida de similitud** con la cual generamos una **matriz de distancias** para poder encontrar en ella los vecinos más cercanos a uno dado, en nuestra implementación quedan recogidas tres tipos de distancia que son:

- **Distancia Euclídea**: Es la medida de distancia más común a utilizar, se deduce del teorema de Pitágoras, calcula la distancia más corta entre dos puntos X y Y de un espacio **n-dimensional**, su fórmula matemática es:

$$d(X,Y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2} \tag{3.1}$$

- **Distancia Manhattan**: Es una distancia relacionada con la distancia euclídea pero en vez de hacer la distancia más corta en diagonal, realiza la medida por *manzanas*, es decir, calcula la distancia entre dos puntos X y Y de un espacio **n-dimensional** en función de cuadrículas equivalentes, su fórmula matemática es:

$$d(X,Y) = \sum_{i=1}^{n}|x_i - y_i| \tag{3.2}$$

- **Distancia Canberra**: Es un medida derivada de la *distancia de Manhattan*, calcula la distancia entre dos puntos X y Y de un espacio **n-dimensional** en función de cuadrículas equivalentes aplicando una ponderación, su fórmula matemática es:

$$d(X,Y) = \sum_{i=1}^{n}\frac{|x_i - y_i|}{|x_i| + |y_i|} \tag{3.3}$$

Estos cálculos de distancias métricas quedan recogidos en el código fuente como **distance.py** lo que lo hace extensible para futuras métricas que se quieran incluir.

3.1.3. Generación de predicciones

Cuando obtenemos los valores futuros observados de cada vecino, es necesario realizar una media ponderada de los mismos para poder obtener una predicción temporal sobre el dato de entrada e instante temporal que estamos evaluando, para ello usamos diferentes **funciones de ponderación** las cuales describiremos a continuación:

- **Ponderación equitativa**: es el método más simple y usado de todos, consiste en darle el mismo *peso* a todos los vecinos de una forma equitativa de la distancia que tengan hacia el conjunto a evaluar, la fórmula de la media aplicando esta ponderación es la siguiente:

$$X = \frac{1}{k}\sum_{i=1}^{k}x_i = \frac{x_1 + x_2 + \cdots + x_k}{k} \tag{3.4}$$

  Donde k es el número de vecinos, X el resultado de la media y x_i corresponde a la predicción de cada vecino i.

- **Ponderación por proximidad**: Este método consiste en darle un peso a la predicción de cada vecino según la distancia a la que se encuentre del conjunto evaluado.
  La fórmula con este método de ponderación es la siguiente:

$$X = \frac{\sum_{i=1}^{k} x_i * \frac{1}{y_i}}{\sum_{i=1}^{k} \frac{1}{y_i}} = \frac{x_1 * \frac{1}{y_1} + x_2 * \frac{1}{y_2} + \cdots + x_k * \frac{1}{y_k}}{\frac{1}{y_1} + \frac{1}{y_2} + \cdots + \frac{1}{y_k}} \tag{3.5}$$

  Siendo k el número de vecinos, X el resultado de la media , x_i la predicción de cada vecino i e y_i la distancia al conjunto evaluado i.

- **Ponderación Lineal**: Esta ponderación aplica un peso a cada elemento en función del orden distancia que tiene, es decir, aplica un mayor peso a los elementos más próximos y menor a los más lejanos sin tener en cuenta la distancia exacta a la que se encuentra, solo el orden de distancia, su fórmula aplicada es la siguiente:

$$X = \frac{\sum_{i=1}^{k} x_i * (k - i + 1)}{\sum_{i=1}^{k} (k - i + 1)} = \frac{x_1 * k + x_2 * k - 1 + \cdots + x_k}{k + k - 1 + \cdots + 1} \tag{3.6}$$

  Donde k es el número de vecinos, X el resultado de la media y x_i corresponde al elemento de predicción de cada vecino i.

Los cálculos de las medias ponderadas quedan recogidos en el código fuente como **weights.py** lo que lo hace extensible para futuras ponderaciones que se quieran incluir.

3.2. Entrenamiento y validación

El algoritmo *k*-NN hace uso del aprendizaje basado en instancias que consiste en entrenar la generación de valores de las nuevas instancias basándose en las ya observadas con anterioridad dividiendo la serie temporal en varias partes. **Normalmente un conjunto de datos se divide en dos partes, entrenamiento y validación**, uno para cada propósito. En series temporales los datos están ordenados temporalmente y eso sucede también con estos conjuntos: el de entrenamiento consiste en un primer tramo de la serie y el de validación en el tramo siguiente. Sin embargo, **el k-NN necesita pasado para buscar instancias y por eso se divide en tres siendo el primer tramo de inicialización**. A continuación hablaremos de cada segmento detenidamente:

- **Inicialización**: Al ser un algoritmo basado en instancias, necesitamos un tramo desde el principio la serie temporal hasta cierto punto que se crea conveniente, estos datos serán un pasado donde podremos buscar los *k* vecinos para predecir instantes temporales, suele ser la fase que más instancias tiene sobre la serie temporal.

- **Entrenamiento**: Consiste en realizar predicciones desde el siguiente instante temporal del segmento de inicialización hasta el principio del segmento de validación de la serie temporal. Trata de encontrar los valores de *k* y de *d* que minimizan el error, para ello se prueba un rango de *k* desde 1 a k_{max} y para *d* desde 1 a d_{max}, al tener que probar todas las combinaciones de *k* y *d* se genera un gran coste en tiempo. Con esto conseguiremos buscar los

parámetros para el algoritmo más **óptimos** a la hora de realizar las predicciones, los cuales serán los que cometan el **menor error posible en las predicciones**. Logramos calcular el error aplicando diferentes **medidas del error**, hablaremos de ellas en el siguiente apartado.

- **Validación**: Una vez obtenemos unos parámetros k y d óptimos para la serie en la que estamos prediciendo, basados en lo aprendido en la **fase de entrenamiento**, se realiza las predicciones para todos instantes temporales desde el siguiente instante temporal a la fase de entrenamiento hasta el final de la serie. Tras conseguir predecir estos instantes temporales, se comparan con los resultados reales de cada instante, consiguiendo un **índice de error** para cada uno de estos instantes temporales.

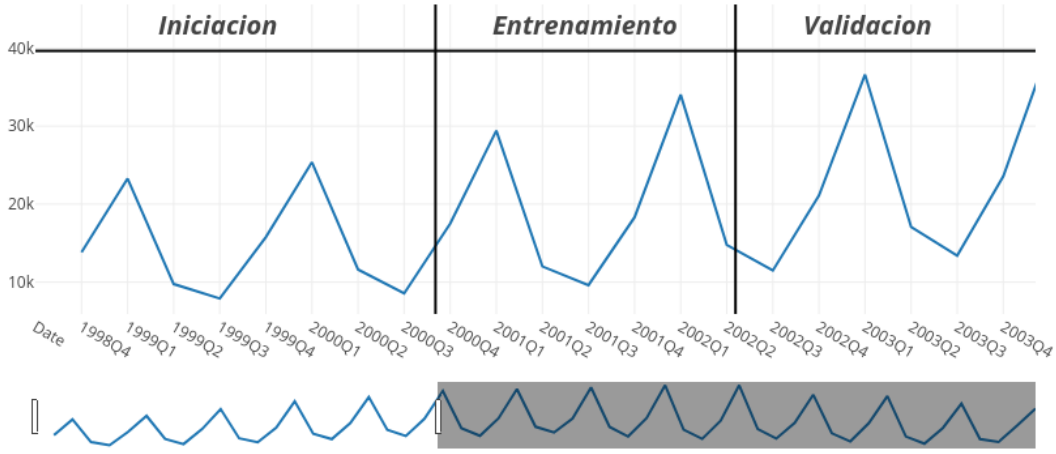

Figura 3.1: División de las diferentes segmentos de una serie temporal

En la *figura 3.1* podemos observar como se dividen las fases del algoritmo a lo largo de la serie temporal, para realizar estas divisiones de la serie, dispondremos del **parámetro *init*** para fijar la instancia temporal donde empezaremos a predecir con el algoritmo k-NN según en que fase nos encontremos.

3.2.1. Medida del error

A la hora de validar las predicciones del algoritmo, es necesario elegir los valores adecuados para las diferentes combinaciones de k y d en la **fase de entrenamiento**. Para poder hacer esto es necesario comprobar los errores entre la predicción obtenida y el valor real en cada instancia de la fase calculando a través de **medidas del error** un valor de error entre el valor real y la predicción. Sea para n el número de elementos, y_i el valor real al instante temporal evaluado i e $\hat{y}_i$ la predicción a ese instante i, se definen las siguientes medidas de error:

- **Error medio (ME)**: Consiste en hacer en hacer una media usando la diferencia entre el valor real y la predicción, es poco aconsejable de utilizar ya que los valores negativos y positivos pueden cancelar unos a otros, su fórmula matemática es:

$$ME = \frac{\sum_{i=1}^{n} y_i - \hat{y}_i}{n} \tag{3.7}$$

- **Error medio absoluto (MAE)**: Está basado en el *error medio*, pero para evitar estas anulaciones en el sumatorio trata los datos en valor absoluto dando un resultado de error más fiable:

$$MAE = \frac{\sum_{i=1}^{n} |y_i - \hat{y}_i|}{n} \tag{3.8}$$

- **Raíz cuadrada del error cuadrático medio (RMSE)**: Es un método para el análisis del error en regresión, penaliza más que el MAE los errores grandes. Es muy utilizado en el análisis de temperatura y clima, su fórmula matemática es la siguiente:

$$RMSE = \sqrt{\frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}} \tag{3.9}$$

- **Error medio porcentual (MPE)**: Se trata de una medida del error en porcentaje de precisión,no debe utilizarse si hay valores reales iguales a 0, como en el *error medio*, pueden darse el casos de que haya valores que se anulen entre sí:

$$MPE = \frac{100\,\%}{n} \sum_{i=1}^{n} \frac{y_i - \hat{y}_i}{y_i} \tag{3.10}$$

- **Error medio porcentual absoluto (MAPE)**: Funciona igual que el *error medio porcentual*, solo que, da un porcentaje más certero al ser los datos en valor absoluto evitando así, la anulación de errores entre sí:

$$MAPE = \frac{100\,\%}{n} \sum_{i=1}^{n} \left| \frac{y_i - \hat{y}_i}{y_i} \right| \tag{3.11}$$

Los cálculos de error métrico quedan recogidos en el código fuente como **error.py** para el sumatorio de valores y en **KNN_Optim.py** para su división, la implementación fue diseñada así, ya que *spark* tiene que realizar primero el sumatorio de diferencias para acto seguido aplicar la operación con el número de datos n correspondiente al *error* usado.

Capítulo 4

Implementación y optimización del algoritmo

La implementación de nuestro *k*-NN en *Apache Spark* queda guardado en el repositorio con la URL: https://github.com/saguila/KNN-clústerComputed/tree/master/TFG protegido bajo una licencia Apache 2.0, en el mismo se encuentra la descripción y funciones comentadas y explicadas para su funcionamiento incluyendo versión de *PySpark* utilizada.
A continuación hablaremos de su implementación del algoritmo *k*-NN tal y como fue descrito en la *sección 3*, las funciones principales requieren siempre como parámetros la serie temporal (*rdd*), los parámetros *d* y *k* y el número de elementos de la serie (*n*) que es calculado usando la función de *Spark count()*, el resto de parámetros tienen definición por defecto luego no son obligatorios para que la función se ejecute [6]. Las funciones que implementan el algoritmo son las siguientes:

- **KNN_Next**: Función que predice el algoritmo *k*-NN para el siguiente instante de la serie temporal.
- **KNN_Optim**: Función que implementa el entrenamiento del algoritmo de para optimizar los parámetros *k* y *d*, usando en un rango determinado, diferentes combinaciones de *k* y *d* .
- **KNN_Past**: Predice valores correspondientes a diferentes fases temporal de la serie.

Para poder hacer lectura de las series temporales volcaremos los datos sobre una estructura de datos llamada RDD (*Resilient Distributed Dataset*) que sirve para poder procesar la serie temporal de **forma distribuida** utilizando *Spark*, se implementan tres pasos del algoritmo usando esta estructura de manera distribuida, estos son :

- La preparación de los datos en función del parámetro *d*
- El cálculo de la matriz de distancias
- La generación de la predicción

4.1. Implementación paralela del algoritmo

En este apartado, hablaremos acerca de las funciones distribuidas utilizadas por todas las funciones principales, explicando el contenido y funcionamiento de cada una de ellas.

4.1.1. Preparación de los datos

Para preparar los datos para comenzar el algoritmo, hay que crear un RDD de la serie temporal tal y como podemos observar en el *cuadro 4.1* que hace referencia una serie temporal con el instante temporal *t* y su valor correspondiente *y*.

$Instante\ temporal$	t_1	t_2	t_3	t_4	$\cdots$	t_{n-2}	t_{n-1}	t_n
$Valor$	y_1	y_2	y_3	y_4	$\cdots$	y_{n-2}	y_{n-1}	y_n

Cuadro 4.1: Mostramos una entrada de un RDD con una serie temporal (*t*) con su valor (*y*)

A continuación, realizamos un proceso agrupación de los datos de la serie tal y como observamos en el *cuadro 4.2*, se agrupan los valores de *d* instancias temporales, fijando para cada fila un identificador para cada agrupación siendo identificador 1 el primer instante de la serie temporal procesada y *n-d* la última instancia de la serie procesada. El valor de predicción de cada agrupación hará correspondencia al siguiente valor de la serie temporal (siendo este y_{d+id}).

$Identificador$	$Valores$						$Predicción$
1	y_1	y_2	$\cdots$	y_{d-2}	y_{d-1}	y_d	y_{d+1}
2	y_2	y_3	$\cdots$	y_{d-1}	y_d	y_{d+1}	y_{d+2}
3	y_3	y_4	$\cdots$	y_d	y_{d+1}	y_{d+2}	y_{d+3}
4	y_4	y_5	$\cdots$	y_{d+1}	y_{d+2}	y_{d+3}	y_{d+4}
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n-d-3$	y_{n-d-3}	y_{n-d-2}	$\cdots$	y_{n-5}	y_{n-4}	y_{n-3}	y_{n-2}
$n-d-2$	y_{n-d-2}	y_{n-d-1}	$\cdots$	y_{n-4}	y_{n-3}	y_{n-2}	y_{n-1}
$n-d-1$	y_{n-d-1}	y_{n-d}	$\cdots$	y_{n-3}	y_{n-2}	y_{n-1}	y_n
$n-d$	y_{n-d}	y_{n-d+1}	$\cdots$	y_{n-2}	y_{n-1}	y_n	$None$

Cuadro 4.2: RDD después de prepararse la serie en función de *d*

Diseñar esta función fue algo complicado ya que *Spark* trabaja de **forma distribuida cada instancia de la serie**, la solución inicial unificar varias copias de la serie temporal igual al valor del parámetro *d* y agruparlas entre sí para conseguir los datos tratados. Esta solución ocupaba una gran cantidad de espacio en memoria acabando por exigir un ***clúster*** más potente de lo necesario para preparar los datos de la serie, además exigía un gran coste en tiempo al tener que redistribuir *d* cada RDD. Buscando alternativas, se optó por realizar una solución proporcionando a cada valor varios identificadores correspondientes a las agrupaciones que pertenecía dicho valor, y acto seguido, agrupar los valores por cada identificador dado quedando como se describió en el *cuadro 4.2*, no exigía necesidad de hacer copias de la serie temporal lo cual permitió reducir el coste en espacio y tiempo considerablemente.

Esta función de preparación de datos se encuentra definida en las función **dRdd** descrita en el fichero fuente **dRdd.py**, recibe por entrada, el RDD que contiene la serie temporal (rdd), el parámetro del algoritmo *d* y el número de instantes que contiene la serie temporal (n), devolviendo como salida la serie temporal agrupada.

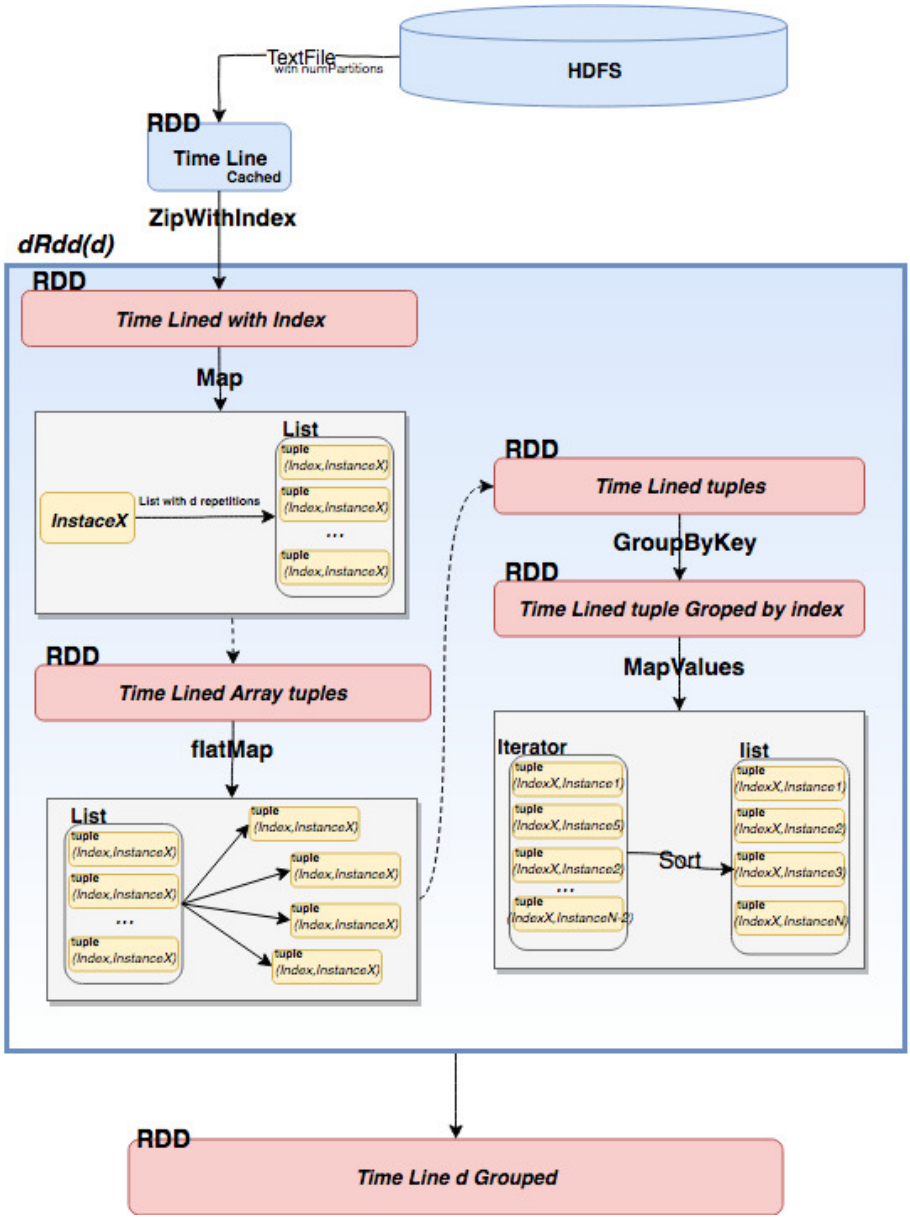

Figura 4.1: Diagrama de la función de agrupación de instantes temporales RDD.

En los diagramas 4.1, 4.5, 4.6 y 4.9 dentro de la función principal se representan en color:

- **Gris** : Para una función determinada.
- **Rojo** : Para un RDD, colección de objetos, que está distribuida en varias máquinas.
- **Morado** : Para variables de Broadcast.
- **Amarillo** : Para variables de ámbito local.

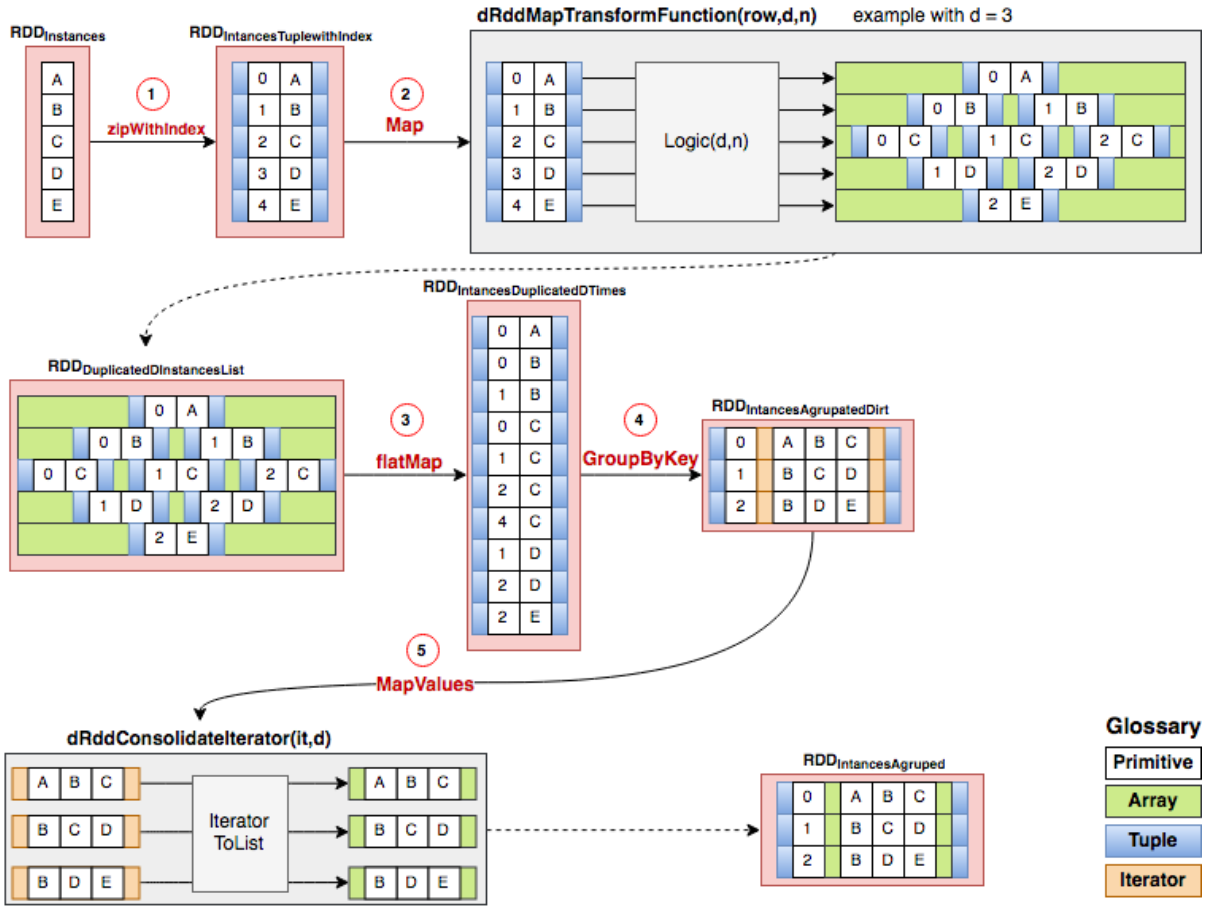

Figura 4.2: Explicación en detalle del algoritmo de agrupación de instancias temporales dRdd.

La *figura* 4.2 representa una ejecución del algoritmo con d=3. A, B, C, D y E son las instancias temporales que se van a agrupar. En el glosario podemos observar los tipos de datos que tenemos en cada uno de los pasos hasta obtener la agrupación temporal. En la figura podemos observar los siguientes pasos:

1. En este paso se realiza un zipWithIndex para poner un índice a las instancias temporales para tener una referencia que se usará posteriormente.

2. Se realiza un Map en el que se le pasa la función dRddMapTransformFunction, se encarga de coger los índices generados en el paso anterior y mediante una lógica crear arrays con las instancias temporales duplicadas necesarias para formar la agrupación.

3. Se separan las instancias duplicadas en vectores pasando a ser tuplas índice-instancia usando una función de flatMap.

4. Se agrupan las instancias temporales por indice usando GroupByKey, esta agrupación nos devuelve los índices y un iterador a las instancias agrupadas, esta estructura no nos interesa, necesitamos tener los datos de una forma directa.

5. Se usa una función de MapValues para trabajar solo con los valores (los iteradores a las instancias) ya que la clave (el índice o referencia temporal) la queremos seguir manteniendo, sobre este MapValues se aplica la funciondRddConsolidateIterator, se encarga de transformar los iteradores a instancias en una lista ordenada con las instancias agrupadas, estas representan la serie temporal agrupada final.

4.1.2. Búsqueda de los *k* vecinos

A la hora de calcular una matriz de distancias, se genera un proceso de coste elevado en tiempo, ya que exige comparar dos series temporales una que contiene los instantes temporales a predecir y otra con la serie temporal completa para poder obtener la distancia de cada instante temporal a predecir con su pasado obteniendo los *k* vecinos más cercanos de cada instancia a predecir.
En un principio se implementó almacenando las dos series en dos RDDs, al calcularse las distancia entre dos series distribuidas en varios ejecutores exigía mucho coste en tiempo, ya que ambos ambas series temporales debían redistribuirse entre los ejecutores varias veces. Este problema llevo a investigar alternativas mejores dando como resultado al uso de las **variables de broadcast** [4] explicadas en el *sección 2.3.3*.

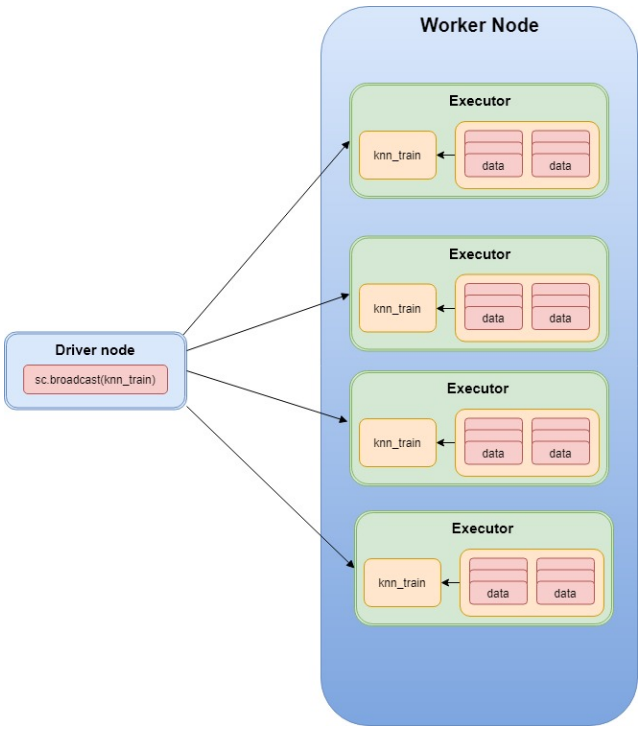

Figura 4.3: Trabajo de *spark* distribuyendo las **variables de broadcast** para generar la matriz de distancia distribuida

Aplicando este método conseguimos mayor velocidad en el algoritmo gracias a que, al tener todos los instantes a predecir en todos los ejecutores, evitamos tener que hacer varias redistribuciones de la serie.

Como observamos en la *figura 4.3*, ganamos velocidad en la ejecución gracias a que todos los ejecutores tienen toda la serie temporal de predicción (knn_train) y la serie temporal completa de forma distribuida (data) que le corresponde a cada ejecutor.
Las variables de *broadcast* tienen un **límite de almacenamiento más pequeño que un RDD** ya que estas no quedan distribuidas, sino que queda almacenado en cada ejecutor todo los instantes temporales de predicción lo cual **genera restricción de espacio en memoria**. Se aconseja para este algoritmo que se utilice $1/3$ de la cantidad de instantes temporales que contenga la serie temporal para realizar las predicciones.

4.1.3. Función de reducción

Después de calcular las distancias a las **variables de broadcast** con la serie temporal distribuida en varios ejecutores obtenemos de forma distribuida una **matriz de distancias**. Cada ejecutor ha obtenido los k vecinos más cercanos a los instantes temporales a predecir **correspondientes al segmento de la serie que tiene almacenado**. Necesitan ser agrupados para poder obtener los vecinos más cercanos a la serie total y poder trabajar con ellos para poder realizar las predicciones. Conseguimos esto **agrupando todos los datos distribuidos en un solo ejecutor**. El problema de esto es que, al **introducir la matriz de distancia en un solo ejecutor puede generar desbordamiento de memoria**, para evitar este caso, ha de aplicarse una **función de reducción** que consiste en **agrupar solo los datos necesarios** de la matriz de distancia, en este caso, los k vecinos más cercanos correspondientes a cada instante temporal a predecir, el resto de datos de la matriz de distancia se eliminan ya que no son necesarios, así el algoritmo gana notablemente una **mejora en coste de espacio**.

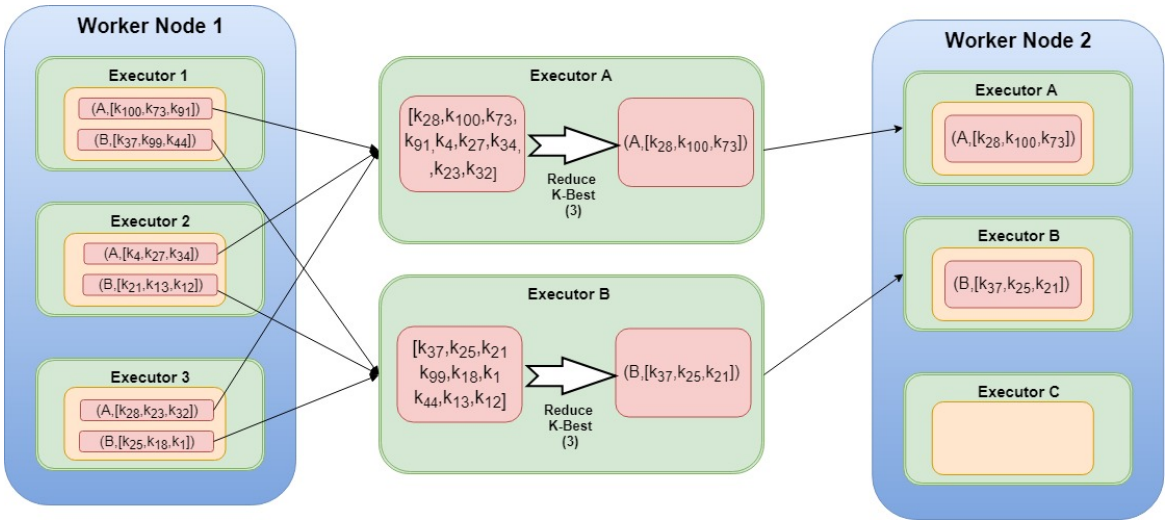

Figura 4.4: Trabajo de *spark* con la función de reducción

Observando la *figura 4.4* comprobamos que cada ejecutor ha calculado la distancia a 2 instantes temporales (instantes A y B). Cada instante temporal corresponde a una tupla tipo clave-valor,

siendo la clave, el identificador al instante temporal a predecir (en este caso A y B) y como valor tiene **un vector ordenado** con los *k* los vecinos más cercanos del pasado al instante temporal a predecir. Teniendo en cuenta que elegimos 3 vecinos, observamos que, cada ejecutor envía sus 3 vecinos más cercanos correspondientes a su segmento de la serie, a un ejecutor disponible, acto seguido, aplicamos la reducción y eliminamos los vecinos que no vamos a utilizar de las diferentes particiones manteniendo ordenados los *k* vecinos del más cercano al más lejano en todo momento. Al final de esta operación se almacena la transformación del RDD con los vecinos más cercanos de cada instante temporal redistribuido entre todos los ejecutores de cada nodo trabajador.

4.2. Funciones principales

En esta sección hablaremos acerca de las funciones principales para entrenar y validar el algoritmo usando comúnmente las funciones distribuidas explicadas en el punto 4.1. Estas funciones contienen un diagrama de flujo explicativo.

4.2.1. KNN_Next

Esta función realiza la predicción para el siguiente instante temporal a la serie. Esta función se encuentra implementada en el fichero **KNN_Next.py**, su descripción es la siguiente:

Algorithm 1 KNN_Next: Predice el siguiente instante temporal a la serie (n+1)

1: **function** *KNN_Next:*$(rdd, d, k, n, distance, weight)$

Entrada: Un RDD con una serie temporal (rdd), los parámetros *d* y *k* del algoritmo, el número de elementos en la serie temporal (n), el tipo de distancia (distance, por defecto distancia euclídea), la ponderación para generar la predicción (weight, por defecto igualdad).

Salida: la predicción al siguiente instante temporal a la serie (n+1).

2: $data \leftarrow dRdd(rdd, d, n)$
3: $knn\_last \leftarrow data.filter(id == n)$
4: $matrix \leftarrow Mdistance(iterator(data), knn\_last, distance, k)$
5: **return** $matrix.reduceByKey().mean(k\text{-}vecinos, weight).collect()$

Para aclarar el funcionamiento de este pseudocódigo, realizaremos una breve explicación y nos apoyaremos en el diagrama de flujo en la *figura 4.5*:

1. Comenzamos por procesar los datos para poder trabajar con ellos en función de *d* **usando la función *dRdd*** (sección 4.1.1) y la almacenamos en una variable *data*, dando a lugar **un RDD con los valores agrupados** (línea 2).

2. Almacenamos el **último instante temporal** en una variable de broadcast (sección 4.1) *knn_last* para poder predecir sobre el último instante de la serie temporal (línea 3).

3. Comparamos todo el pasado de la serie con la variable broadcast de forma distribuida y **elegimos los *k* vecinos más cercanos al último instante temporal en cada ejecutor** (sección 4.1, línea 4).

4. Utilizamos la **función reducción** (sección 4.1.3) para ordenar y obtener los *k* vecinos distribuidos entre los ejecutores eliminando los datos no utilizados, tras esto, aplicamos una **media ponderada** a sus valores para generar la predicción (línea 5).

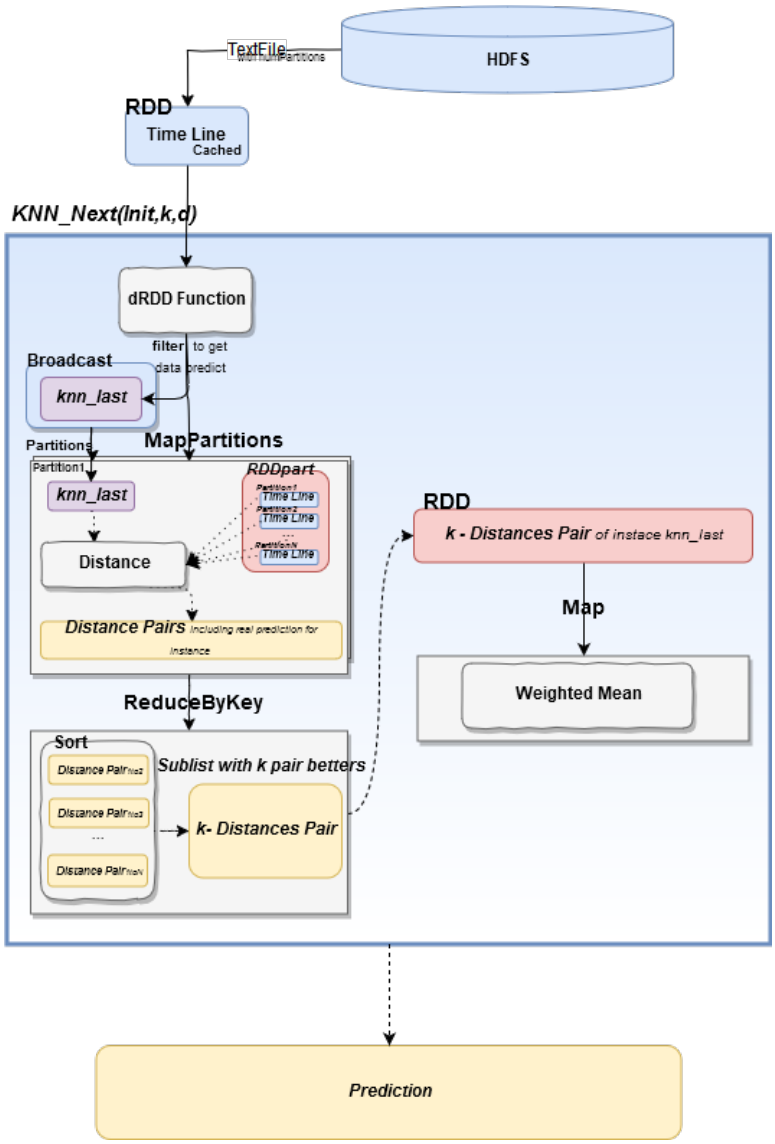

Figura 4.5: Diagrama de flujo KNN_Next

Esta función se diseñó como la función base del algoritmo, con ella se consiguió una **derivación del resto de funciones principales**, se fue alterando consiguiendo optimizarlo gracias al uso de **variables de broadcast**.

4.2.2. KNN-Past

Esta función se encarga de realizar la **validación** de las predicciones a partir de una serie temporal con los parámetros *k* y *d* optimizados y el parámetro *init* recortando la serie hasta el siguiente instante al segmento de entrenamiento y así generar un RDD en el cual podremos comparar con los datos reales de la serie con las predicciones obtenidas dando lugar a un RDD que contenga el error entre la predicción y la realidad de cada instante temporal predicho. Como hemos comentado, antes de pasar por esta función se necesita **optimizar los parámetros** *k* y *d* antes de utilizarse a través la función **KNN_Optim**.

Algorithm 2 KNN_Past: Genera predicciones para instantes temporales ya conocidos

1: **function** *KNN_Past:*$(rdd, d, k, n, distance, init, weight)$

Entrada: Un RDD con una serie temporal (rdd), los parámetros *d* y *k* del algoritmo, el número de elementos en la serie temporal (n), el tipo de distancia (distance, por defecto distancia euclídea),el parámetro init (init, por defecto ($\frac{2n}{3}$), la ponderación para generar la predicción (weight, por defecto igualdad).

Salida: Una serie temporal entrenada.

2: $data \leftarrow dRdd(rdd, d, n)$
3: $knn\_train \leftarrow data.filter(t > init - i - 1$ ***and*** $t \neq n - d)$
4: $matrix \leftarrow Mdistance(iterator(data), knn\_last, distance, k)$
5: **return** $matrix.reduceByKey().mean(k\text{-}vecinos, weight)$

Para comprender mejor el pseudocodigo vamos a explicar brevemente cada punto para dejar más claro el funcionamiento apoyándonos en diagrama de flujo de al *figura 4.6*.

1. Comenzamos por procesar los datos para poder trabajar con ellos en función de *d* en usando la función *dRdd* y la almacenamos en una variable *data*, dando a lugar un RDD con los valores agrupados (línea 2).

2. Almacenamos todos los instantes temporales conocidos que tengan el identificador superior al parámetro *init* en variables de broadcast (*knn_train*) para poder predecir sobre estos instantes (línea 3).

3. Comparamos todo el pasado de la serie con las variables de broadcast de forma distribuida, cada ejecutor elegirá los *k* vecinos más cercanos de forma ordenada a cada instante temporal del fragmento de la serie que contenga el ejecutor (línea 4).

4. Utilizamos la **función reducción** para ordenar y obtener los *k* vecinos distribuidos entre los ejecutores eliminando los datos no utilizados, tras esto, aplicamos una **media ponderada** a sus valores para generar la predicción para cada uno de los instantes temporales a predecir (línea 5).

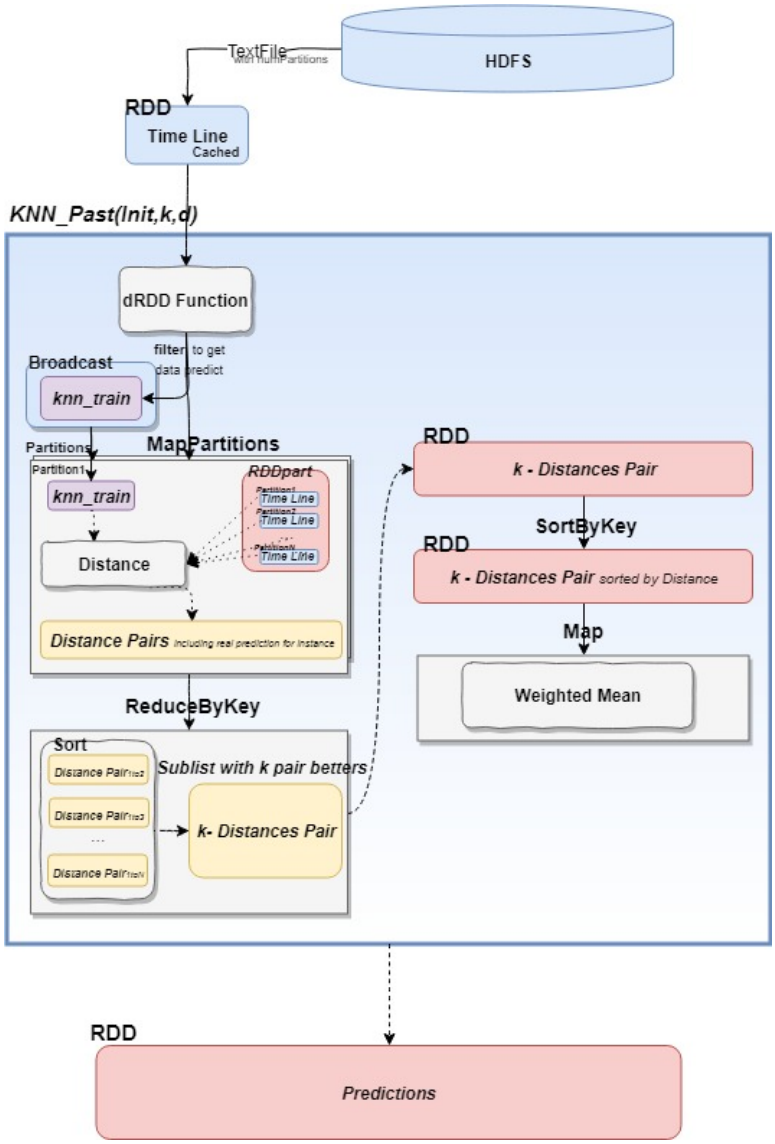

Figura 4.6: Diagrama de flujo KNN_Past

4.2.3. KNN_Optim

Como hemos comentado anteriormente, antes de poder realizar cualquier predicción es necesario seleccionar unos parámetros k y d con el **menor error** en la predicción posible.
Para poder encontrar **los mejores parámetros posibles** se utiliza **un rango de pruebas para los parámetros *k* y *d***, probándose con las diferentes **combinaciones aplicando el algoritmo *k*-NN** para cada combinación con un *init* fijado como **fase de entrenamiento**. Este proceso requiere mucho coste en tiempo, a continuación explicaremos la implementación de esta función:

Algorithm 3 KNN_Optim: busca los mejores valores en un rango fijado para k y d

1: **function** *KNN_Optim:*$(rdd, d, k, n, distance, init, weight, err)$

Entrada: Un RDD con una serie temporal (rdd), los rangos máximos para las combinaciones de los parámetros *d (1:d)* y *k (1:k)* del algoritmo, el número de elementos en la serie temporal (n), el tipo de distancia (distance, por defecto distancia euclídea),el parámetro init (init, por defecto ($\frac{2n}{3}$), la ponderación para generar la predicción (weight, por defecto igualdad) y la medida de error utilizada para la optimización (err, por defecto MAE).

Salida: Una tupla con tres valores que son: (error mínimo, *k* óptimo, *d* óptimo)

```
2:  optimData ← False
3:  for i ← 1 to d do
4:      data ← dRdd(rdd, i, n)
5:      knn_train ← data.filter(t > init − i − 1 and t ≠ n − d)
6:      matrix ← Mdistance(iterator(data), knn_last, distance, k)
7:      groupMatrix ← matrix.reduceByKey()
8:      for j ← 1 to k do
9:          check ← groupMatrix.map(error(mean(j-vecinos, weigth), y, err))j, i)
10:         if optimData == False or optimData.Error > check.Error then
11:             optimData ← check
12:
13: return optimData
```

Este pseudocodigo es el más complejo de todas las funciones, consta de dos bucle anidados, uno externo y otro interno para probar las combinaciones de *k* y *d* [1].
A continuación explicaremos el pseudocodigo detenidamente teniendo el cuenta el diagrama de flujo de la *figura 4.9*.

1. Se inicializa la variable *optimData* a False para tener un indicador de que no hay parámetros óptimos seleccionados (línea 2).

2. Abrimos el primer bucle externo hasta *d* (línea 3). Este bucle realiza la siguiente secuencia podemos observar su diagrama en la *figura 4.7*.

   2.1 Comenzamos por procesar los datos para poder trabajar con ellos en función de *d* en usando la función *dRdd* y la almacenamos en una variable *data* , dando a lugar un RDD con los valores agrupados (línea 4).

   2.2 Almacenamos todos los instantes temporales conocidos que tengan el identificador superior al parámetro *init* en variables de broadcast (*knn_train*) para poder predecir sobre estos instantes (línea 5).

   2.3 Comparamos todo el pasado de la serie con las variables de broadcast de forma distribuida y elegimos los *k* vecinos más cercanos de forma ordenada a cada instante temporal que contenga cada ejecutor (línea 6).

   2.4 Utilizamos la **función reducción** para ordenar y obtener los *k* vecinos distribuidos entre los ejecutores eliminando los datos no utilizados (línea 7)

2.5 Abrimos el bucle anidado hasta *k* (línea 8) y volvemos a empezar el proceso hasta realizar estos pasos *d* veces, una vez realizados devolveremos la variable **optimData** que almacena el menor error encontrado en las combinaciones, la *k* y la *d* óptimas en una tupla de tres datos.

.

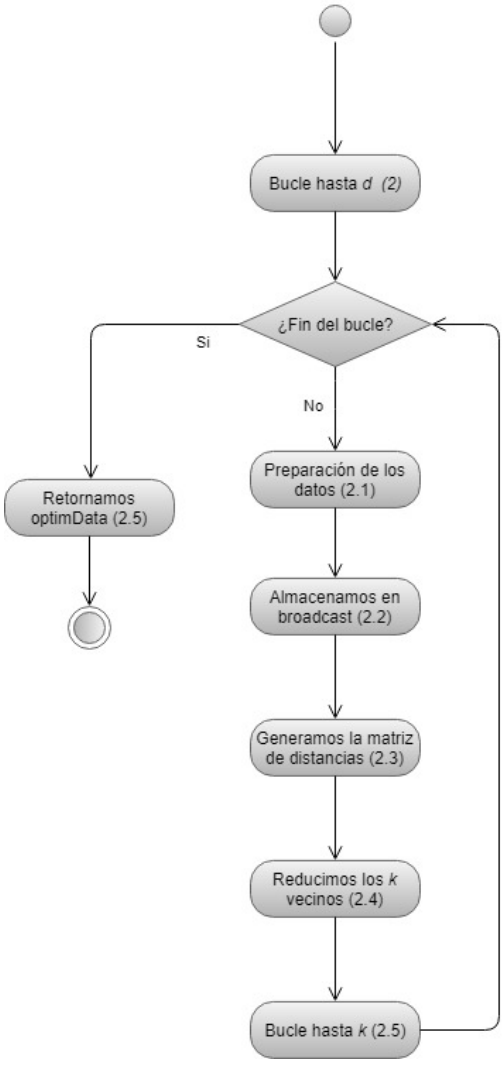

Figura 4.7: Trabajo del bucle externo

Ahora vamos a explicar el funcionamiento del bucle anidado hasta *k*, este bucle **se utiliza para realizar las combinaciones con la *d* actual con todos los rangos de *k* posibles buscando el menor error posible**, así evitamos que rehacer más veces de las necesarias la matriz de distancia, que es el proceso más costoso del algoritmo.
Podemos observar en la *figura 4.8* el esquema de funcionamiento del bucle interno, a continuación explicamos paso a paso el funcionamiento del bucle:

1. Abrimos un bucle anidado hasta *k* (línea 8).

2. Realizamos la predicción de los *j* vecinos más cercanos y calculamos el error usando un tipo de **medida de error**. Se almacena en una tupla de tres valores para comprobar si es el menor error que llamaremos *check* con los valores: error a comprobar, valor de *k* actual (*j*) y valor *d* actual (*i*).

3. Comprobamos si el error en la tupla *check* tiene **menor error** que la tupla óptima elegida actualmente, de ser así, almacenamos *check* en *optimData*.

4. Repetimos este proceso hasta probar todas los valores fijados en el rango de *k*.

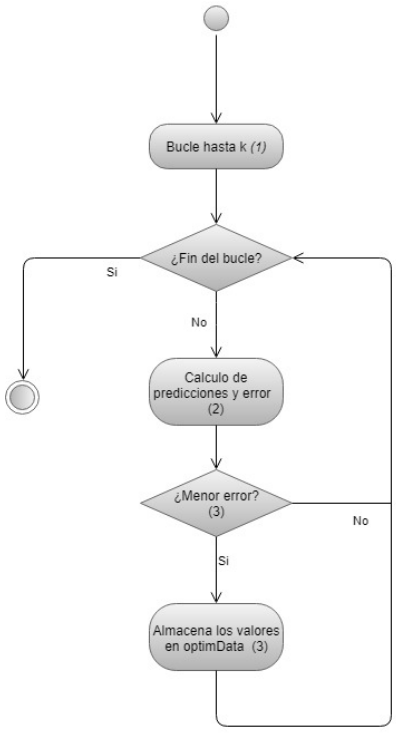

Figura 4.8: Trabajo del bucle interno

Como podremos observar en la explicación esta función la consideraremos crítica ya que, según aumentan las combinaciones a analizar de *k* y *d*, siendo *d* la que aumentará drásticamente el coste en tiempo del algoritmo.

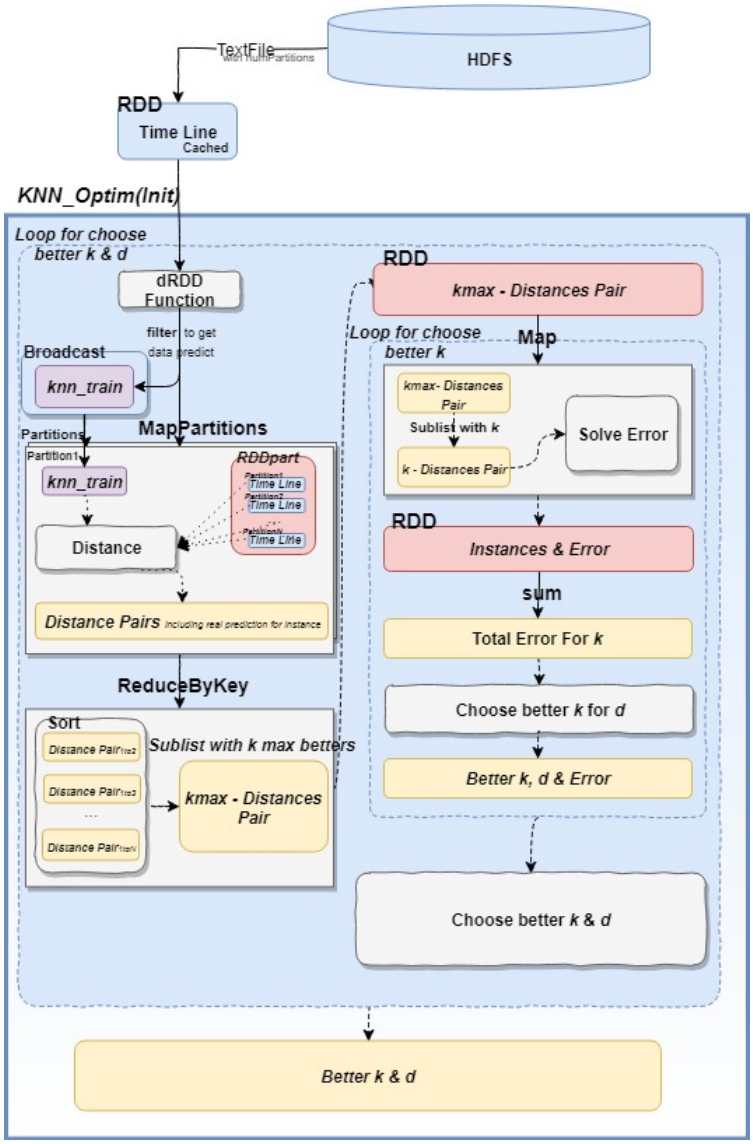

Figura 4.9: Diagrama de flujo KNN_Optim

Capítulo 5

Experimentos y comparación con la implementación mono-máquina

A la hora de realizar predicciones temporales podemos encontrarnos diferentes **tamaños** de series temporales lo cual conlleva a diferentes conclusiones de que método usar, *distribuido* o *mono-máquina*. En este capítulo hablaremos y analizaremos la eficiencia de cada método según el tamaño de datos de la **serie temporal**.

5.1. Series temporales consideradas

En este apartado presentaremos las series temporales que hemos usado para probar el funcionamiento del algoritmo.

5.1.1. Bitcoin Historical Data

Contiene un histórico por minuto del precio de la cripto moneda Bitcoin en Dólares desde enero de 2012 hasta julio de 2018 con un total de 3.255.876 registros.
https://www.kaggle.com/mczielinski/bitcoin-historical-data/home.
La predicción de los valores futuros de *Bitcoin* con la solución implementada presenta un problema grave, debido a que es una serie con tendencia estrictamente creciente a la que no se la puede aplicar directamente el algoritmo, puesto que no se observan patrones ni buenos vecinos para usar en las predicciones.
Con el fin de hacerla más útil se ha hecho una transformación, filtrando los registros pertenecientes al periodo inicial en el que la criptomoneda era estable quedando un total de 2.188.132 empezando desde julio del 2012 hasta julio de 2018. Además, se ha transformado la serie y se ha convertido en la serie de los rendimientos logarítmicos del valor del bitcoin.
Con esto hemos conseguido transformar un problema de predicción del próximo valor del *Bitcoin* a un problema de predicción de periodos de rendimientos haciendo útil la aplicación del algoritmo.

5.1.2. Sunspot Daily

Contiene un histórico con las observaciones diarias del número de manchas solares desde el 1 de enero de 1818 hasta el 30 Junio de 2018 con un total de 73.230 registros.

http://www.sidc.be/silso/datafiles.
Esta serie al contrario que la anterior, directamente se pueden observar patrones, no ha necesitado ninguna transformación de los datos para hacer predicciones útiles con el algoritmo, lo único que se ha realizado fue una adaptación a la entrada esperada por el programa, eliminando campos que no pertenecen a la serie temporal.

5.1.3. Sunspot Monthly

Igual que el anterior contiene un histórico con las observaciones diarias del número de manchas solares pero con una periodicidad mensual desde el enero de 1 de enero de 1749 hasta Junio de 2018 con un total de 3.233 registros.
http://www.sidc.be/silso/datafiles.
Como paso con el conjunto de datos con periodicidad diaria, se ha realizado fue una adaptación a la entrada esperada por el programa, eliminando campos que no pertenecen a la serie temporal.

5.2. Alcance de la solución mono-máquina en R vs solución Spark

La solución implementada por los compañeros en *R* para una máquina [1] esta limitada por la cantidad de memoria que ocupa la matriz de distancias que se genera para resolver el algoritmo. Esta matriz con una cantidad de registros determinada puede superar la capacidad de una máquina, debido a que cualquier computador de 64 bits como mucho puede referenciar 8 Tb en memoria ram. Con una máquina con características similares a la que se ha usado en el desarrollo (véase *sección 2.4.1*), se pueden procesar unos 44000 instantes temporales como máximo que ocupan 7,3 Gb de memoria ram. El *clúster* donde se realizarán las pruebas para la solución de *Spark* con 4 máquinas serán tal y como se describe en la *sección 2.4.2*.

5.3. Escenarios

Para poder analizar la eficiencia entre entre el *algoritmo distribuido* contra *mono-máquina* usaremos diferentes **históricos** según su tamaño y analizaremos las funciones una a una para realizar una comparación de coste en tiempo entre ambos, todos los históricos nombrados y funciones a probar en este punto se encuentran almacenados en el repositorio de GitHub de la *sección 5.1*.

5.3.1. Serie pequeña

En estas pruebas utilizaremos principalmente el histórico **Sunspot.monthly** y un parámetro *init* de 400 datos, luego realizaremos la predicción sobre 2.833 . A continuación mostraremos cada una de las funciones y su coste en tiempo del algoritmo.

KNN_Next

$Spark$				
d \ k	10^0	10^1	10^2	10^3
10^0	2,11	1,95	1,76	1,93
10^1	1,6	1,71	1,73	1,53
10^2	2,15	2,17	1,9	1,83
10^3	4,69	4,99	4,38	5,5

Cuadro 5.1: Coste en tiempo de la aplicación con **Spark** en segundos para KNN_Next

R				
d \ k	10^0	10^1	10^2	10^3
10^0	0,03	0,03	0,03	0,03
10^1	0,05	0,05	0,05	0,05
10^2	0,31	0,32	0,32	0,33
10^3	7,1	8,17	8,31	8,57

Cuadro 5.2: Coste en tiempo de la aplicación con **R** en segundos para KNN_Next

Observando los resultados se puede ver que claramente que la programación *mono-máquina* es más rápida cuando hay parámetros menores, según va creciendo (d) va aumentando el coste progresivamente llegando a un punto en el que es mejor hacer **programación distribuida**.
El valor de (k) no ha tenido apenas influencia luego podemos deducir que el número de vecinos prácticamente, **no afecta al algoritmo**.

KNN_Past

$Spark$				
d \ k	10^0	10^1	10^2	10^3
10^0	8,89	9,37	13,38	17,34
10^1	27,45	27,87	27,89	27,91
10^2	133,7	141,29	143,91	147,24
10^3	950,02	851,11	919,04	829,14

Cuadro 5.3: Coste en tiempo de la aplicación con *Spark* en segundos para KNN_Past

R				
d \ k	10^0	10^1	10^2	10^3
10^0	0,25	0,31	0,25	0,32
10^1	0,27	0,29	0,32	0,30
10^2	0,57	0,54	0,61	0,55
10^3	7,14	7,16	7,16	7,25

Cuadro 5.4: Coste en tiempo de la aplicación con *R* en segundos para KNN_Past

Tras realizar las pruebas en ambas máquinas, observamos que *spark* empeora dramáticamente según van aumentando el retardo (d), es bastante peor en tiempo en comparación con *R*. Pero hay que tener en cuenta que las soluciones distribuidas son mucho más costosas y contamos con un *clúster* de prueba no productivo, por lo tanto este retardo se puede evitar con un *clúster* más potente ya que podríamos emplear más ejecutores para el algoritmo.

KNN_Optim

$Spark$			
d \ k	10^0	10^1	10^2
10^0	9,24	8,7	27,24
10^1	167,19	343,8	337,28
10^2	8135,74	8635,23	9038,16

Cuadro 5.5: Coste en tiempo de la aplicación con *Spark* en segundos para KNN_Optim

R			
d \ k	10^0	10^1	10^2
10^0	0,30	0,45	3,79
10^1	2,39	5,69	40,49
10^2	59,83	136,10	448,6

Cuadro 5.6: Coste en tiempo de la aplicación con *R* en segundos para KNN_Optim

Al igual que pasaba con el KNN_Past los tiempos son peores, tenemos el mismo problema el *clúster* de pruebas es demasiado sencillo, la complejidad y el tamaño de los datos merma los tiempos.

5.3.2. Serie media

Para este apartado se utilizo una serie temporal de pruebas **sunspot.daily** descrito en el punto 5.1.2.

Probamos a ejecutarlo varias veces con diferentes configuraciones, usando *Amazon Elastic MapReduce* con un clúster con 5 máquinas m5.xlarge workers con 4 cores y 16 gigas por máquina que hacen un total de 80 gigas y 20 cores. tras ejecutar todas las funciones con un valor de 3 para los parámetros *d*, *k* y un *init* de 48820, el algoritmo tardo 4.369 segundos (1 hora y 12 minutos aproximadamente) en ejecutarse usando las tres funciones distribuyendo los tiempos así:

KNN_Optim	52 min
KNN_Past	21 min
KNN_Next	10 seg

Cuadro 5.7: Coste en tiempo de la aplicación con *Spark* para cada función

Se ha realizado una sola ejecución de todas las funciones debido a los costes del alquiler del *clustér* los cuales encarecen su precio por horas de uso.
Tras finalizar las pruebas con *Spark*, probamos a ejecutar en *mono-máquina* con la misma serie temporal dando como resultado un fallo de ejecución debido a que la matriz de distancia excedía el límite de memoria máxima (7.3 GB), se amplio el límite y se volvió a probar dando lugar a un "Screen Freeze"del ordenador utilizado, luego no se consiguió probar. Al no poder realizar la prueba sobre *mono-máquina*, no se pudo deducir una mejora o empeoramiento en coste en tiempo. Como no se pudo demostrar la velocidad se realizó un recorte al histórico para que fuera aceptable usando una mono máquina demostrando que, al haber usado un *clúster* con mayor capacidad de memoria que la *mono-máquina* se consiguió una predicción más certera como podemos ver en los resultados:

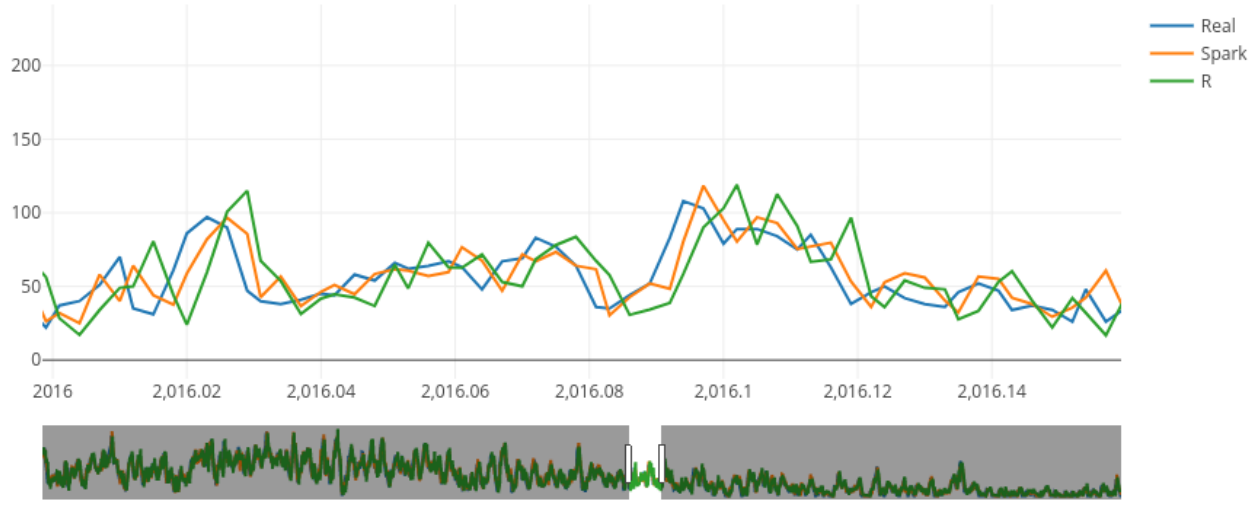

Figura 5.1: Diferencia entre predicción y realidad para un fragmento de 2000 instantes temporales.

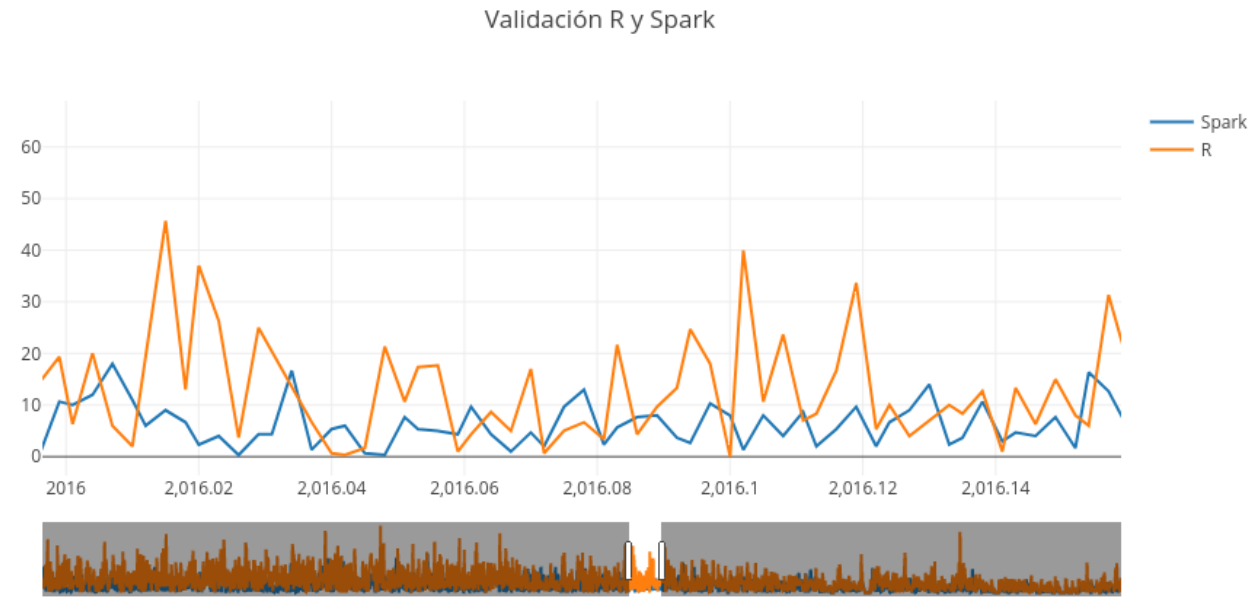

Figura 5.2: Validación del error absoluto entre *Spark* y *R* con 9 combinaciones.

Para estas pruebas hemos probado el histórico completo en *Spark* y los últimos 12000 instantes temporales del histórico para *R* prediciendo 2000 instantes temporales tanto para entrenamiento como para validación, utilizando el error métrico MAE y un valor 3 para los parámetros k y d dando un total de 9 combinaciones para la búsqueda de los parámetros óptimos.

MAE	$Train$	$Test$
R	10,575	8,139
$Spark$	7,387	6,558

Cuadro 5.8: Diferencia de error con MAE entre *R* y *Spark*

En la *figura 5.1* podemos observar la diferencia entre los datos predichos por cada método, se ve una similitud no muy clara entre la realidad y las predicciones lo cual ha requerido comprobar con una **validación del error**, en las predicciones que podemos observar en la *figura 5.2* y el *cuadro 5.8* que demuestran que, al tener un histórico con mayor cantidad de instantes temporales, *Spark* ha realizado unas predicciones más certeras que las realizadas en *R*.

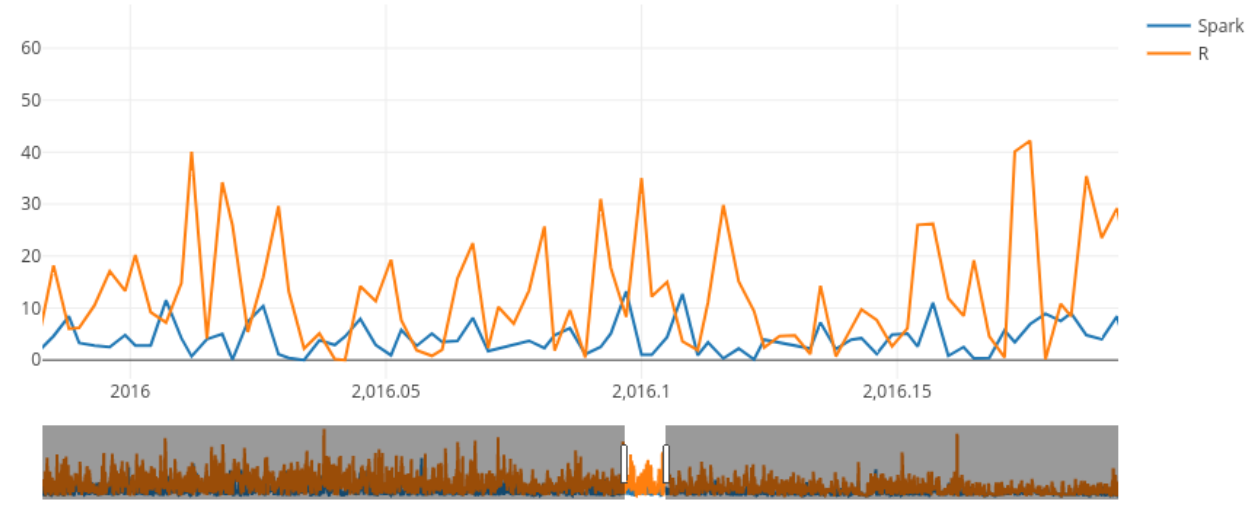

Figura 5.3: Validación del error absoluto entre *Spark* y *R* con 100 combinaciones

Se repitieron las mismas pruebas alterando los parámetros k y d a 10 dando un total de 100 combinaciones para buscar los parámetros óptimos. Tal y como observamos en la *figura 5.3* con los que pudieron comprobar la importancia de aumentar las combinaciones en el entrenamiento, observamos que *Spark* sigue siendo más precisa al disponer de un histórico mayor, pero ambas

predicciones mejoraron con el aumento de combinaciones.
Observando la *figura 5.4* se ve claramente la mejoría de usar mayores combinaciones para el entrenamiento, en este caso con *Spark*.

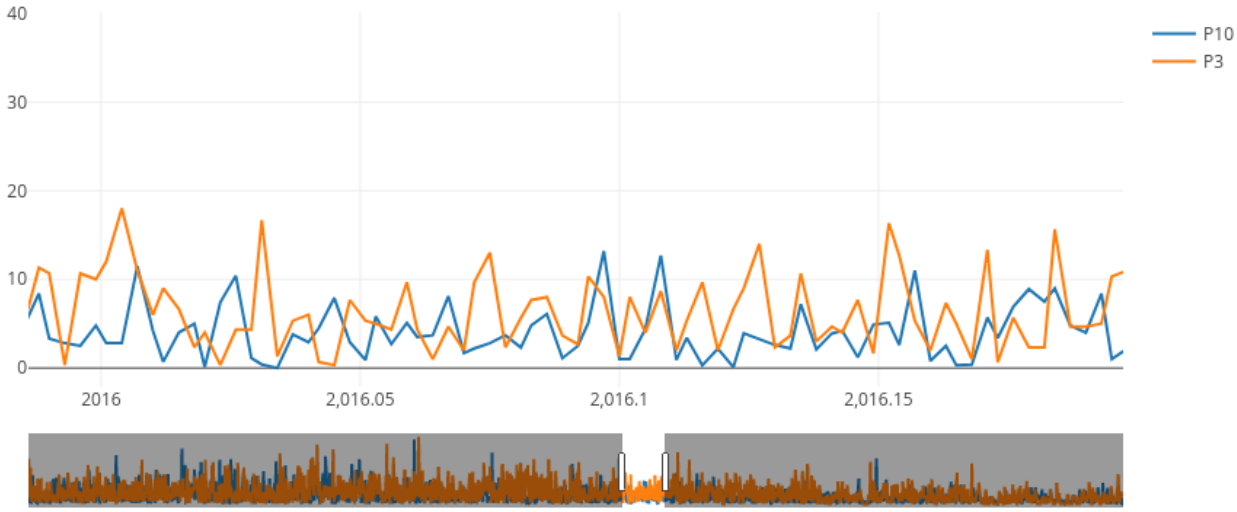

Figura 5.4: Error absoluto de *Spark* en 100 (P10) contra 9 (P3) combinaciones

5.3.3. Serie Grande

Se intentó ejecutar en el *clúster* de *Amazon Elastic MapReduce* con la misma configuración de *clúster* que se realizó para la serie temporal **bitcoinUSD.csv** correspondiente a Bitcoin Historical descrito en el punto 5.1.1 con **rendimientos logarítmicos aplicados**, el cual se quedo en ejecución durante 3 horas, al ser un coste económico excesivo se canceló. La ejecución se realizó en nuestro *clúster* aplicando una ejecución con 32 ejecutores distribuidos en los 4 nodos usando como parámetros k y d a 3 y con un *init* fijado para predecir 10.000 instantes temporales. En el caso la *mono-máquina*, al ser un histórico de más de 3 millones de datos, como era obvio la máquina no fue capaz de ejecutarlo por el exceso de memoria que exigía (30 terabytes de memoria aproximadamente). Se redujo el histórico a un a los últimos 30.000 instantes temporales con los mismos parámetros de predicción. Estos fueron sus resultados:

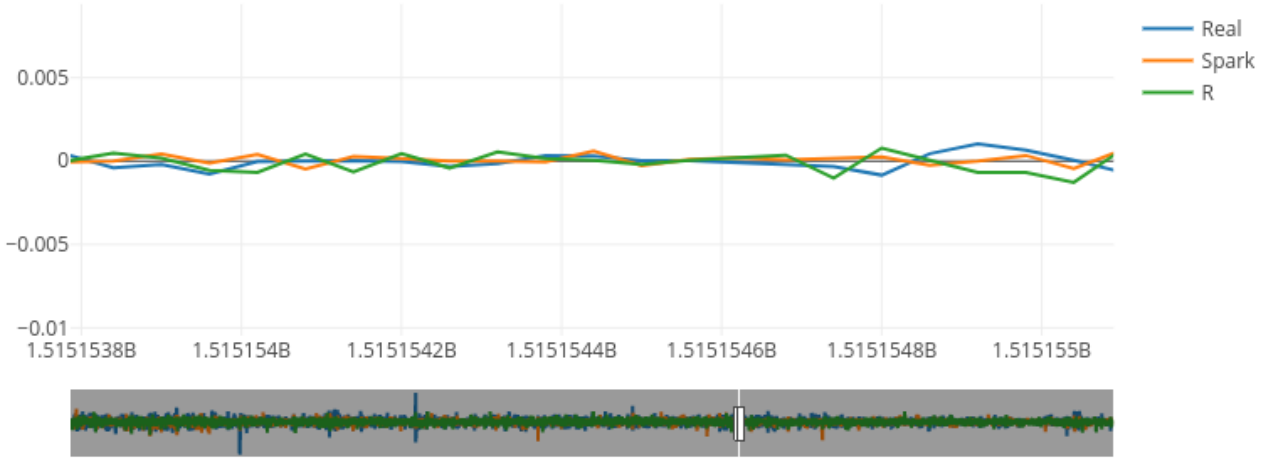

Figura 5.5: Diferencia entre predicción y realidad para un fragmento de 10.000 instantes.

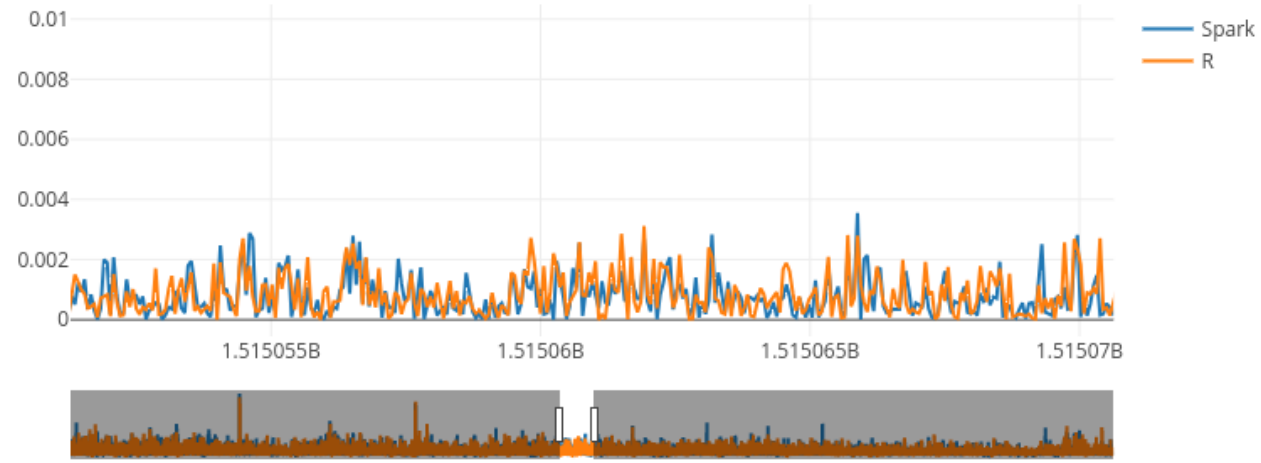

Figura 5.6: Fragmento de error absoluto en *Spark* para 10.000 predicciones.

Como podemos observar en la *figura 5.5* los resultados de la predicción han sido muy similares entre *R* y *Spark* han sido muy similares pero a pesar de ello, *Spark* al tener un histórico para la predicción más grande que *R* ha conseguido predicciones un poco mejores. Al observar la *figura 5.6* viendo la diferencia de error, de esto podemos confirmar que, **a mayor histórico mejores predicciones** se realizan.

5.4. Conclusiones

Tras realizar las pruebas usando programación distribuida y *mono-máquina* podemos deducir que la programación *mono-máquina* es una clara ganadora en lo que se refiere a coste en tiempo en series temporales pequeñas, en cambio, la programación distribuida usando un *clúster*, destaca cuando tenemos que realizar predicciones temporales de gran tamaño, llevando una gran cantidad de tiempo realizar las predicciones temporales pero, siendo capaz de **almacenar y predecir sobre cantidades inmensas de datos**.
Si contáramos con una cantidad de recursos ilimitada, el algoritmo distribuido es mejor opción, pero si lo que queremos hacer es un análisis con un coste bajo sobre un conjunto de datos pequeño la solución *mono-máquina* es la mejor alternativa. Con las pruebas que hemos ido realizando con el serie temporal de hemos llegado a la conclusión que a partir de serie temporal sunspot.daily de tamaño medio, un *clúster* con 80 gigas y 20 cores con la solución distribuida daría un mejor resultado que una máquina de 16 gigas y 4 cores con solución *mono-máquina*.

Capítulo 6

Trabajo Futuro

Este proyecto de PySpark ha sido diseñado con el fin de poder ser **mantenido** y poder añadir nuevas funcionalidades fácilmente.
Se han deducido las siguientes futuras funcionalidades:

- **Agregación de nuevas métricas**: Como hemos hablado en la *sección 3* existen métricas de **distancia, ponderación y error** para la implementación del algoritmo. Como se ha comentado en los apartados 3.1.1, 3.1.2 y 3.2.1, cada medida o ponderación está implementada en un fichero aparte y podrían añadirse nuevas métricas a cada fichero para dar más opciones a la hora de realizar **predicciones temporales**. *parámetros* al algoritmo fácilmente.

- **Añadir utilidades de lectura en la entrada de datos** que permitan soportar ficheros con múltiples formatos, así como poder configurar distintos separadores o la columna con la serie.

- **Implementar un entrenador de particiones**: Dependiendo del número de ejecutores y del tamaño de nuestra serie temporal podemos implementar un entrenador que elija automáticamente el número en el cual particionaremos la serie temporal (particiones de un RDD). Para ello fijaríamos el entrenador a un número exacto de particiones igual al $n^{o}ejecutores*$ $n^{o}cores$ en función de probar a hacer operaciones que requieran la agrupación de los datos (count, reduce, sum...) eligiendo el numero de particiones que menos tiempo necesite para realizar esta operación.

Bibliografía

[1] DANIEL FRANCISCO BASTARRICA LACALLE, JAVIER BERDECIO TRIGUEROS, *Librería para la predicción usando el k-NN: paralelización y visualización de resultados, Universidad Complutense de Madrid, 2018.*

[2] JAVIER ARROYO GALLARDO. *PhD - Métodos de prediccion para intervalos e histogramas,Universidad Pontificia Comillas, 2008.*

[3] JAVIER ARROYO GALLARDO. *Articulo: k-NN para series temporales, 2009.*

[4] JESUS MAILLO, SERGIO RAMÍREZ, ISAAC TRIGUERO, FRANCISCO HERRERA, *kNN-IS: An Iterative Spark-based design of the k-Nearest Neighbors classifier for big data, 2016.*

[5] APACHE HADOOP WEBSITE, *https://hadoop.apache.org/*

[6] APACHE SPARK WEBSITE, *https://spark.apache.org/*

[7] BILL CHAMBERS, MATEI ZAHARIA, *Spark The Definitive Guide, 2017*

[8] BORIS LUBLINSKY, KEVIN T. SMITH, ALEXEY YAKUBOVICH, *Hadoop soluciones Big Data, 2013*.

Agradecimiento

A los tutores del proyecto *Javier Arroyo Gallardo* y a *Albert Meco Alías*, por darnos la oportunidad de participar en este proyecto, por la guía y el apoyo recibido.
A *CoreNetworks S.L* por facilitarnos las maquinas usadas para el desarrollo y las pruebas del TFG.
A *Daniel Francisco Bastarrica Lacalle* y a *Javier Berdecio Trigueros*, por su trabajo que nos ha servido de apoyo y comparativa a nuestro proyecto.